Universität Augsburg
Institut für Informatik
Programmierung verteilter Systeme

Konzepte und Techniken im Organic Computing

WS 2007/2008

Seminarband

Bernhard Bauer
Holger Kasinger
Raphael Romeikat
Viviane Schöbel

Vorwort

Im Wintersemester 2007/2008 wurde von der Gruppe *Programmierung verteilter Systeme* der Universität Augsburg das Seminar 'Konzepte und Techniken im Organic Computing' veranstaltet. Dieser Seminarband umfasst alle studentischen Arbeiten, die im Rahmen des Seminars erstellt wurden.

Den Beginn des Bandes bildet die Arbeit von *Christoph Stempfle*, welcher eine umfassende Einführung in das Forschungsgebiet 'Organic Computing' (OC) bietet. Im Anschluss daran stellt *Bekr Turki Hatam* in seiner Arbeit Rules als eine erste, grundlegende Technik zur Regelung des lokalen Verhaltens von OC Systemen vor. Diese Technik ist im Kontext des OC eng mit dem Begriff der Policies verbunden, einer Technik zur Steuerung des globalen Verhaltens von OC Systemen. Letztere betrachtet *Andreas Meixner* in seiner Arbeit genauer.

Nikolai Klimov widmet sich in seiner Arbeit der Tatsache, dass in modernen Rechnern sowie in verteilten Systemen viele Vorgänge parallel ausgeführt werden, welche zum besseren Verständnis für den Entwickler, Benutzer oder Administrator mit geeigneten Sprachen modelliert werden müssen. Da in verteilten, dezentralen Systemen darüberhinaus die Koordination einzelner Entitäten eine wichtige Rolle spielt, zeigt *Michael Boegler* in seiner Arbeit verschiedene dezentrale Mechanismen auf, welche diese Koordination ermöglichen und unterstützen. Anschließend befasst sich *Martin Burkhard* mit verschiedenen Design Patterns für autonome, dezentrale Systeme.

Andrey Ostroverkhov hingegen befasst sich in seiner Arbeit mit den Vor- und Nachteilen dieser selbst-organisierenden emergenten OC Systeme, insbesondere mit dem Auffinden und der Kontrolle von ungewolltem, emergenten Fehlverhalten darin. Den letzten Beitrag zu diesem Band liefert abschließend *Michael Buthut*, indem er neun ausgewählte Forschungsprojekte der Phase I des Schwerpunktprogramms Organic Computing (SPP 1183) genauer betrachtet und damit einen Einblick in die aktuelle Forschung in diesem Gebiet bietet.

Die in diesem Seminarband enthaltenen Arbeiten spiegeln natürlich nur einen Teil aller im Organic Computing vorhandenen und eingesetzten Techniken wieder. Aufgrund des Einfluss' verschiedener Forschungsgebiete auf OC gibt es eine weitaus größere Vielfalt bekannter Techniken, welche sich in der ein oder anderen Form immer wiederfinden lassen. Eine spannende Frage für die Zukunft des Organic Computing bleibt jedoch, ob sich alle diese Konzepte und Techniken so zusammenbringen lassen, dass es eines Tages gelingen wird, vertrauenswürdige, vom Nutzer akzeptierte OC Systeme zu entwickeln, zu betreiben, zu warten und zu recyceln.

Februar 2008 — Die Editoren

Inhaltsverzeichnis

Einführung in Organic Computing

Christoph Stempfle

Universität Augsburg
Christoph.Josef.Stempfle@student.uni-augsburg.de

Zusammenfassung In dieser Arbeit wird zunächst auf die Notwendigkeit eingegangen, bestimmte Prozesse innerhalb von komplexen System zu automatisieren. Um dies zu erreichen, kann man auf die Prinzipien des Autonomic und Organic Computing zurückgreifen. Es wird zu beidem eine Einführung geboten, die auf die Geschichte, Eigenschaften und typische Architekturtypen eingeht, welche auch anhand konkreter Beispiele erläutert werden. Somit soll das Verständins geweckt werden, dass zukünftige Systeme immer mehr autonom agieren werden und müssen und die Prinzipien des Organic Computing eine Möglichkeit darstellen, das Komplexitätsproblem besser beherrschbar zu machen.

1 Einleitung

In den letzten 15 Jahren haben Computersysteme weltweit zunehmend an Komplexität gewonnen. Konnte man früher seine Server noch alle per Hand verwalten und konfigurieren, so ist dies heute nur noch in bedingtem Maße möglich. Dafür sind die Abläufe, die in großen Netzwerken stattfinden, zu komplex. Ein weiterer Nachteil der Wartung durch menschliche Administratoren entsteht auch bei hoch sicherheitskritischen Anwendungen, sei es in der Industrie, beispielsweise in einem Kernkraftwerk, oder aber auch in eingebetteten Systemen, wie Flugzeugen oder Autos. In solchen Systemen wäre es fatal, wenn plötzlich Kernfunktionen ausfielen, ohne dass man sich um die Reparatur dieser kümmert, zumal der Pilot oder Fahrer die internen Systemprozesse nicht überwachen könnte. Selbst wenn er es könnte, wäre es nicht unbedingt sinnvoll. Er müsste womöglich in kürzester Zeit richtig auf hochkomplexe Vorgänge reagieren, wobei ihm natürlich wahrscheinlich Fehler unterlaufen würden. Deswegen wäre es gut, solche Wartungs- und Reparaturprozesse in irgendeiner Form zu automatisieren. Im Folgenden werden verschiedene Möglichkeiten dazu aufgezeigt: Autonomic Computing, der von IBM entwickelte Ansatz, und Organic Computing, das sich innerhalb der deutschen Forschergemeinde durchgesetzt hat.

2 Autonomic Computing

Der erste Ansatz zur Lösung von Problemen in komplexen Systemen wurde von IBM aus der Natur abgeschaut und unter dem Namen „Autonomic Computing“ vermarktet. Im folgenden wird eine Übersicht über die Eigenschaften und die Architektur autonomer Computersysteme gegeben.

2.1 Definition

Paul Horn von der Firma IBM prägte den Begriff des „Autonomic Computing“. Er definierte autonome Systeme folgendermaßen: „Computer Systems that can regulate themselves much in the same way as our autonomic nervous system regulates and protects our bodies.“ [1, S.25] Man versucht die oben beschriebene Problematik nun nicht mehr mit zusätzlicher Rechenpower in den Griff zu bekommen, sondern durch gezieltes Sammeln und Einsetzen von Daten. Nach Auswertung dieser Daten kann das System dann automatisch entsprechend, wenn möglich auch richtig, reagieren. IBM nennt diesen Ansatz Autonomic Computing. Ein autonomes Computersystem hat folgende Eigenschaften zu erfüllen: Es sollte

- flexibel
- immer erreichbar und
- transparent sein. [5]

Unter flexibel versteht man, dass die zu sammelnden Daten von überall her kommen können. Transparenz bedeutet in diesem Zusammenhang, dass das System zuverlässig arbeitet, obwohl der User von der innernen Struktur des Systems nichts weiß. Natürlich war es ein weiteres Erfordernis, dass die neue Technologie eine deutliche Kostenersparnis zu älteren Ansätzen bieten musste. Zudem musste ein autonomes System, trotz seiner Autonomie, weiterhin zuverlässig funktionieren. Wenn erstmal die Grundfunktionalität gewährleistet und erforscht worden war, sollte mit diesem Mechanismus aber auch eine Zusammenarbeit von verschiedensten Komponenten auf der ganzen Welt besser funktionieren. Durch Selbstkonfiguration sollte Rechenpower sinnvoll gebündelt werden, um große Simulationen, wie z.B. Wettervorhersagen durchzuführen.

2.2 Eigenschaften

Um dieses ehrgeizige Ziel zu erreichen, definierte IBM acht Elemente, die gründsätzlich immer erfüllt sein müssen, unabhängig des weiteren technischen Fortschritts auf diesem Gebiet.

2.2.1 Die Acht Elemente

1. Das autonome System muss sich selbst und alle ihm zugehörigen Komponenten kennen.
2. Aus diesem Grund muss es sich auch an gewissen Standards orientieren, um das Management heterogener Netzwerke auch zu gewährleisten.
3. Die Komplexität sollte nach Außen hin verborgen bleiben.
4. Es muss sich automatisch und dynamisch rekonfigurieren können.
5. Es gibt keinen Zustand, an dem es sich nicht verändert. Es sucht ständig nach Verbesserungen in der Konfiguartion, je nachdem, welches Optimierungskriterium man ihm vorgibt.

6. Es muss sich natürlich aber auch selber schützen können, zum Beispiel gegen Angriffe von Außen.
7. Außerdem muss es sich auch selbstständig anpassen können, wenn neue Komponenten in das System eingebaut werden.
8. Ein autonomes System sollte sich selbst heilen können, so sollte es sich beispielsweise nach Abstürzen wieder selbst starten können. [4]

2.2.2 Self-X-Eigenschaften Aus diesen Anforderungen heraus ergaben sich die berühmten Self-X-Eigenschaften:

– Selbstkonfigurierend: Ein System mit dieser Eigenschaft erlaubt es, sich während des Betriebs umzukonfigurieren und das mit minimaler Interaktion mit dem Menschen. Somit sind Veränderungen in der Infrastruktur des Systems, wie das Hinzufügen neuer Komponenten, nicht länger ein Problem. Dies kann natürlich die Produktivität erheblich steigern, da nun viel weniger Konfigurationsaufwand seitens der Administratoren erfolgen muss und das erweiterte System schneller einsatzfähig ist.
– Selbstoptimierend: Hier kommt es darauf an, die zur Verfügung stehenden Ressourcen möglichst gut auszunutzen. Da dies oft ein hochkomplexer Vorgang ist, wäre es schön, diese Verantwortlichkeit von der Administration in großen Teilen auf das System selbst zu übertragen, da ihm hierfür bei geschickter Datenhaltung alle notwendingen Informationen vorliegen. Als kurzfristiges Ziel in der Entwicklung des Autonomic Computing wird nur die Überwachung und Verwaltung der Performance des Systems angestrebt. Mittel- bis langfristig könnte dieses Ziel spezieller werden: das System könnte aus alten Erfahrungen lernen und seine Performance auf ganz spezielle Businessprozesse hin ausrichten.
– Selbstschützend: Das Ziel des Selbstschutzes ist, dem richtigen Benutzer die richtige Information zur richtigen Zeit zur Verfügung zu stellen. Dies wird über vordefinierte Policies, dem System bekannte Regeln und Richtlinien, an das es sich halten muss, und über verschiedene Rollen, die die Benutzer kategorisieren, gesteuert. Ein selbstschützendes System ist in der Lage, das Eindringen von Fremden zu bemerken und selbstständig dagegen vorzugehen, um den Schaden jeglicher Attacken so gering wie möglich zu halten. Ein Vorteil ist es, dass es sich hierbei um einen Prozess handelt, der durchgängig stattfindet und somit den Administratoren viel Zeit einspart. Außerdem ist es auch zuverlässiger und billiger als das Überwachen des Systems durch den Menschen.
– Selbstheilend: Diese Systeme können Fehlerzustände von einzelnen Komponenten selbst entdecken und diese auch wieder beheben, ohne dass bei der weiteren Verwendung von Grundfunktionalitäten auf anderen Teilen des Systems in irgendeiner Form Einschränkungen entstehen. Somit wird die IT-Umgebung als Ganzes robuster, da keinerlei Bezug zwischen den verschiedenen Komponenten vorhanden ist. [6]

2.3 Weg zum vollständig autonomen System

Es scheint unmöglich, alle diese Ziele auf einmal erreichen zu können. Deshalb hielt IBM eine schrittweise Entwicklung dieses komplexen Selbstmanagementprozesses als angebracht. Unter Selbstmanagement versteht man in diesem Zusammenhang die Selbstverwaltung eines Systems, welche ohne menschliches Eingreifen funktioniert. [21]
IBM unterteilte deshalb die Leistungsfähigkeit eines autonomen Systems in fünf Klassen.

1. Die Klasse „Basic“: Zu dieser Klasse gehören heute immer noch die meisten Systeme. Sie weisen keinerlei Dynamik auf und müssen von hochspezialisierten Experten gewartet, kontrolliert, verändert, aufgesetzt und betrieben werden.
2. Systeme, die zur Gruppe „Managed“ gehören, bündeln Informationen auf geschickte Weise, so dass ein geringerer Aufwand zur Administration entsteht.
3. Als „Predictive“ bezeichnet man Infrastrukturen, die bestimmte, typische Zustände des Systems erkennen und den Administratoren auf Grund von Erfahrungen, also abgespeicherten Daten, Empfehlungen zum weiteren Vorgehen aussprechen. Somit wird das Personal, das für die Verwaltung des Sytems zuständig ist, erheblich entlastet. Außerdem können Entscheidungen damit viel schneller und effizienter getroffen und die Änderungen umgesetzt werden.
4. In der Stufe „Adaptive“ vergleicht das System seinen Zustand ständig mit sogenannten Service Level Agreements (SLAs) und leitet anhand dieser geeignete Maßnahmen ein. Somit findet nur noch eine geringe Mensch-Maschine-Interaktion statt. Solche Systeme kann man schon als agil und robust bezeichnen.
5. Der höchste Level in diesem Modell nennt sich „Autonomic“. Solche Systeme sind höchst dynamisch. Um dies zu gewährleisten, müssen Businessregeln und Policies aufgestellt werden, anhand derer das System weiß, wie es im Weiteren vorgehen soll. Die IT Abteilung kann sich nun, anstatt ständig ein Auge auf das System an sich werfen zu müssen, wirklich auf Geschäftprozesse konzentrieren. [3]

2.4 Architekturtyp MAPE

Komplexe Systeme müssen im Vorhinein geplant werden, um eine Umsetzung und Implementierung überhaupt möglich zu machen. Dabei spielt der Architekturtyp eine zentrale Rolle. Unter Architektur versteht man in der Informatik „im Allgemeinen das Zusammenspiel der Komponenten eines komplexen Systems.“ [17]
Ein beliebter Architekturtyp in autonomen System ist das sogenannte Monitor-Analyze-Plan-Execute, oder kurz MAPE, welches von IBM entwickelt wurde. Typischerweise bestehen autonome Systeme aus autonomen Elementen, wie in

Abbildung 1 dargestellt. Diese sind wiederum eigenständige Subsysteme, die entweder Ressourcen beinhalten oder Dienste anbieten. Ein autonomes Element besteht typischerweise wiederum aus einem oder mehreren „managed elements", die an einen einzelnen autonomen Manager gekoppelt sind. Auf einer niedrigen Abstraktionsebene kann es sich bei einem Element um eine Hardware Ressource, wie Festplatte, Prozessor oder Drucker handeln, aber auch um Software, wie eine Datenbank. Ein Element kann aber auch auf einer höheren Ebene angesiedelt sein, also eine ganze Applikation umfassen.[12, S.31,37]

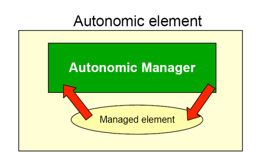


Abbildung 1. Autonomes Element
[12, S.31]

Der autonome Manager ist das Herzstück der MAPE Architektur. Er ist für die Automatisierung der Funktionen zuständig, die das System „self-managing" machen. Die vier Hauptfunktionen Monitor, Analyse, Planung und Ausführung werden im Folgenden erläutert und in Abbildung 2 skizziert.

– Monitor: Er sammelt und verarbeitet Informationen von gemanageten Ressourcen. Diese Informationen bekommt er über eine Schnittstelle von einem Sensor. Er tut dies solange, bis ein Symptom auftritt, das analysiert werden soll. Symptome werden vom Monitor erzeugt, indem dieser gesammelte Daten miteinander in Beziehung bringt und verarbeitet. Dabei weisen Symptome auf den Zustand einer oder mehrerer gemanageter Ressourcen hin.
– Analyse: Diese Funktion stellt Mechanismen zur Verfügung, die das überwachen und analysieren von Symptomen übernehmen. Außerdem stellt die Analyse fest, ob Änderungen erforderlich sind. Wenn ja, wird ein Änderungswunsch an die Planung geschickt. Die Änderung wird aber weiterhin von der Analyse überwacht. Sie stellt auch fest, ob die der Planung vorgeschlagene Änderung die zu Beginn aufgetretenen Symptome behandelt hat.
– Planung: Diese Komponente ist dafür verantwortlich, eine geeignete Prozedur auszuwählen oder zu kreieren, welche gemanagte Elemente und Ressourcen verändert. Nach der Erstellung eines Änderungsplans wird dieser an die Ausführungsfunktion weitergeleitet. Solch ein Plan kann im einfachsten Fall eine einzige Anweisung enthalten, in komplexen Systemen aber auch dutzende Befehle, die mehrere hundert Ressourcen verändern.
– Ausführung (Execute): Sie ist zuständig für die eigentliche Durchführung der Änderung und deren Scheduling. Diese einzelne Stufe ist vor allem wieder in großen Systemen notwendig, da eine Änderung des Gesamtzustands des Systems wiederum aus vielen kleinen Anpassungen besteht, welche alle

einem Kommando im Ausführungsplan entsprechen. Die Ausführungsfunktion gibt die Befehle an sogenannte Effektoren weiter, die für die letztliche (physikalische) Ausführung der Vorgänge zuständig sind. Eine weitere Aufgabe der Ausführungseinheit ist die Aktualisierung der Wissensdatenbank, auf die der autonome Manager zurückgreift, um Elemente zu verwalten.[2, S.18-26] [12, S.32-36]

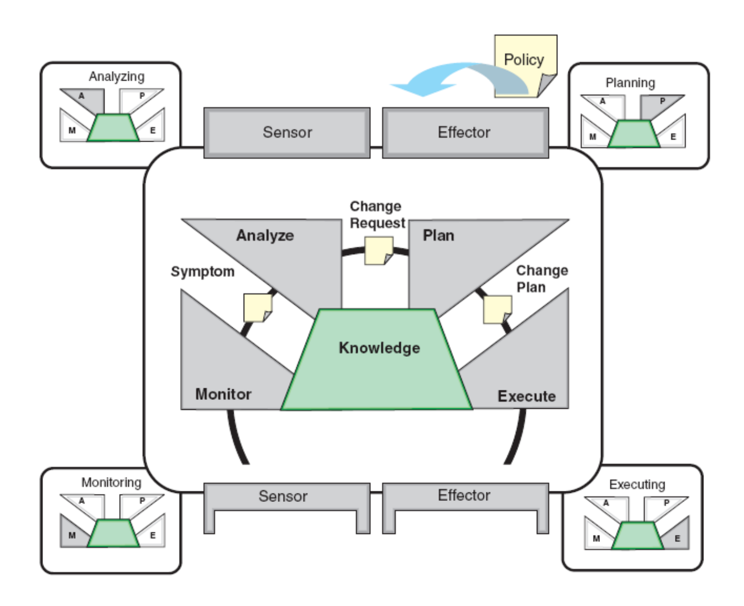


Abbildung 2. Die MAPE Architektur
[2, S.19]

Bei MAPE handelt es sich um ein hybrides Observer-Controller System, wobei der Monitor und die Analyse Teil des Observers sind, die Planung und Ausführung hingegen Teil des Controllers. Durch die Überwachung der Änderungen könnte man die Analyse jedoch auch zu einem Teil dem Controller zuschreiben. Das Wissen, auf das der autonome Manager zurückgreift, ist im Observer und Controller vorhanden. Ein Unterschied zum später beschriebenen klassischen Observer-Controller System ist, dass es keine explizite Kommunikation mit der Außenwelt gibt, das Ganze findet immer innerhalb des Systems statt. Die Aspekte, nach denen ein MAPE System überwacht werden soll, werden in der Planung und Ausführung spezifiziert.[12, S.38]

3 Organic Computing

In diesem Abschnitt wird zuerst auf den Begriff Organic Computing und dessen noch junge Geschichte eingegangen. Es wird im Folgenden erklärt, welche

Gemeinsamkeiten und Unterschiede zum Autonomic Computing existieren. Außerdem wird ein Beispiel für eine konkrete Anwendung der dem Organic Computing zugrunde liegenden Prinzipien gegeben, nämlich die Evolutionsstrategien. Der zweite und dritte Abschnitt des Kapitels befassen sich mit wichtigen Schlagwörtern des Organic Computing, erklären diese und betteen sie in den Gesamtkontext ein. Zu diesen gehören Stichwörter wie Selbstorganisation und Emergenz, welche in 3.2 und 3.3 erklärt werden. In 3.4 werden dann verschiedene Architekturtypen vorgestellt und anhand von praktischen Umsetzungen auch erläutert.

3.1 Geschichte

Im Jahre 2002 fand ein Workshop zum Thema Zukunftsthemen der technischen Informatik statt. Das Ziel dieses Workshops war es, Ergebnisse, die im Laufe des Jahres auf verschiedenen Tagungen erzielt wurden, noch einmal zusammenzufassen. Hierbei bildete sich der Begriff Organic Computing allmählich heraus. Pioniere auf diesem Gebiet waren zu dieser Zeit Prof. Dr.-Ing. Christian Müller-Schloer, Prof. Dr. Theo Ungerer und Prof. Dr. Hartmut Schmeck. Ihr Ziel war es, aus genauen Beobachtungen der Natur Mechanismen zu identifizieren, die auch im IT-Umfeld Anwendung finden könnten. Unabhängig hiervon, war es Christoph von der Malsburg, der schon ein Jahr zuvor, im November 2001, einen Workshop unter dem Namen „Organic Computing“ veranstaltete. Er versteht aber den Begriff des Organic Computings anders: Für ihn bedeutet Organic Computing eine Kombination aus Neurowissenschaften, Molekularbiologie und Software. Sein Ziel ist es, neuartige rechnende Systeme zu entwickeln, die biologische Prinzipien ausnutzen. Den Forschungsaufwand hierfür sieht er allerdings im zeitlichen Rahmen von 10 bis 15 Jahren. [9, S.3] Im Folgenden wird unter Organic Computing der erste Sachverhalt verstanden.
Organic Computing ist prinzipiell eng verwandt mit dem von IBM propagierten Autonomic Computing. Es kann auch als deutscher Ansatz auf diesem Forschungsgebiet verstanden werden. „Ein 'organischer Computer' ist definiert als ein selbstorganisierendes System, das sich den jeweiligen Umgebungsbedürfnissen anpasst. Organische Computer sind selbst-konfigurierend, selbst-optimierend, selbstheilend und selbstschützend.“ [10, S.332] An dieser Definition erkennt man schon Gemeinsamkeiten und Unterschiede zum Autonomic Computing. Gemeinsamkeiten sind die Ausnutzung der Selbst-x-Eigentschaften, wie Selbstkonfiguration, Selbstoptimierung, Selbstheilung und Selbstschutz. Hinzu kommt noch das Phänomen der Selbstorganisation, das sich das Organic Computing Forschungsteam aus der Natur abgeschaut hat. Als Unterschied könnte man anführen, dass der Fokus des Organic Computing eher im verteilten Umfeld und bei eingebetteten Systemen liegt, wohingegen das Autonomic Computing das Selbstmanagement von Servern als Ziel hat. Ein an der Universität Augsburg realisiertes Projekt im Bereich Organic Computing ist der Prototyp der sogenannten „Smart Doorplates“. „Die „Smart Doorplates“, intelligente Türschilder, zeigen eine neue Vision für das Bürogebäude der Zukunft auf. Die Tüschilder stellen aktuelle Informationen über Büroinhaber dar, können in deren Abwesenheit bestimmte

Handlungen übernehmen und weisen Besuchern den Weg zum gegenwärtigen Aufenthaltsort des Büroinhabers über ein Location-Tracking System.“ [16] Nun aber wollen wir die graue Theorie und die reinen Definitionen verlassen und ein praktisches Beispiel anführen, das zum Feld Organic Computing gehört.
Als eines der herausragenden Prinzipien der Natur lässt ich die von Darwin aufgestellte Evolutionstheorie sehen. Seiner Meinung nach überlebt nur immer der am besten Angepasste. [18] Dazu müssen natürlich alle Individuen irgendwie voneinander unterscheidbar sein, also unterschiedliche Eigenschaften besitzen. Bei der Fortpflanzug kommt es dann zu Variationen im Erbgut, die wiederum an die Nachkommen weitergegeben werden. Somit entstehen wieder Individuen mit unterschiedlichen Merkmalen. Bei der Fortpflanzung kommt es meistens auch zu einer Überproduktion, das bedeutet, es gibt mehr Individuen in der neuen Generation, als benötigt werden. Deshalb muss ein Selektionsprozess stattfinden, der nach bestimmten Kriterien, wie Umwelt und Nahrungsangebot die Besten einer Generation auswählt.
Dieses Verfahren hat Prof. Der. Ingo Rechenberg schon in den 70er Jahren versucht, mathematisch zu formulieren und es in der Technik anwendbar zu machen. Die untenstehende Abbildung 3 zeigt anhand eines einfachen Beispiels Rechenbergs Idde. Sein Ansatz war es, einen Vektor frei zu wählen und zuerst ein Duplikat von diesem zu erzeugen. Das Duplikat wurde dann an den einzelnen Komponenten mutiert. Die Mutation an sich ist zufällig und soll hier nicht genauer betrachtet werden. Nun wird der Ursprungsvektor, der sogenannte Elter, und das mutierte Duplikat, das sogenannte Kind, mit Hilfe einer vorher festgelegeten Qualitätsfunktion überprüft. Die Qualitätsfunktion versucht, die Güte beider Vektoren in Bezug auf eine Funktion zu vergleichen und den Besseren zu bestimmen. Nach diesem Ranking der Vektoren wird der bessere ausgewählt und das Verfahren beginnt wieder von vorne. Wie man hier gut erkennt, ist der Algorithmus wirklich der Natur nachempfunden. Sowohl Variation (Mutation) als auch Selektion sind hier beobachtbar. Rechenberg nannte dieses Verfahren die (1+1)-Evolutionsstrategie. Später verfeinerte er das Verfahren immer weiter, so dass es noch allgemeiner wurde. Man kann in der allgemeinsten Form sogar bestimmte Nischen einrichten, in denen Individuen in Bezug auf die Selektion anders behandelt werden, als außerhalb. Somit hatte er ein Werkzeug geschaffen, um Individuen, je nach Beschaffenheit, gezielter behandeln zu können. Er könnte also Vektoren, von denen er weiß, dass sie bezüglich der Qualitätsfunktion schon sehr hochwertig sind, isolieren und sie anders behandeln als neue, zufällig generierte. Die genauere Behandlung würde aber hier zu weit führen. [15]

3.2 Selbstorganisation

Ein wichtiger Begriff der bei Organic Computing Systemen immer wieder auftaucht, ist jener der Selbstorganisation. Da die Prinzipien des Organic Computing oft in verteilten, eingebetteten System vorkommen, ist Selbstorganisation ein nahezu unumgängliches Konzept, um den enormen Anforderungen solcher gerecht zu werden. Auf der einen Seite sind Systeme dieser Art unheimlich komplex, was die Anzahl der Knoten angeht, die in einem solchen System überwacht und

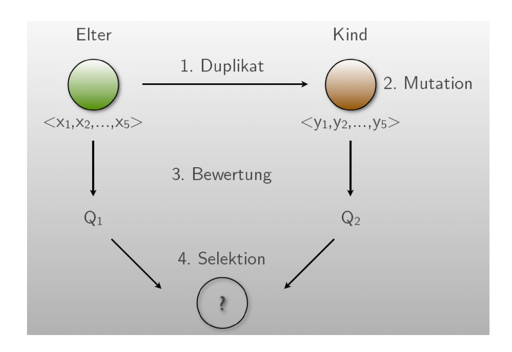


Abbildung 3. Die (1+1)-ES von Rechenberg [15]

korrekt verwaltet werden müssen, andererseits sollten „gute“ Systeme natürlich auch flexibel gegenüber Änderungen der Konfiguration sein. Ein menschlicher Administrator könnte damit vollkommen überfordert sein. Somit ist es für Organic Computing Systeme unbedingt notwendig ist, sich selbst zu organisieren, also ohne Fremdeinwirkung und Einwirkung von Außen.

3.2.1 Definition Aber, was ist Selbstorganisation eigentlich genau? Für die einen ist es ein Begriff, der aus der Systemtheorie stammt, der „eine Form der Systementwicklung bezeichnet, bei der die formgebenden, gestaltenden und beschränkenden Einflüsse von den Elementen des sich organisierenden Systems selbst ausgehen.“ [20] Selbstorganisation ist aber mittlerweile auch in anderen Domänen zu einem wichtigen Schlagwort geworden, sei es in den Naturwissenschaften, der Physik, Biologie und Chemie, oder auch in sozialen und ökonomischen Systemen. Eine andere mögliche Definition betont die Abwesenheit von äußeren Kontroll- und Manipulationsmechanismen, die dazu führt, dass Systeme, die sich in einem unausgewogenen Zustand befinden, Strukturen und Möglichkeiten zu finden, dies zu regulieren. [13]
Eine weitere, sehr genaue, Definition findet man auch im Web: Hier wird der Mechanismus der Selbstorganisation genauer beschrieben, in dem die Komponenten eines Systems untereinander kommunizieren, worauf die Umgebung keinen Einfluss hat. [7]

3.2.2 Eigenschaften Trotz der verschiedenen Sichtweisen in Bezug auf die Definition eines selbstorganisierenden Systems, sind Systemen dieser Art fast immer vier Merkmale eigen:

1. Komplexität: Diese wird erreicht, wenn Komponenten innerhalb eines Systems miteinander in Beziehung stehen, diese Beziehung sich aber jederzeit wieder ändern kann, auf Grund veränderter Anforderungen an das System. Beispielsweise könnte sich eine Komponente im System ändern. Somit müssten alle anderen Knoten, mit denen diese Komponente auf welche Weise

auch immer verbunden ist, jegliche Verweise auf die veränderte Komponente updaten, da diese eventuell ganz andere Dienste zur Verfügung stellt, als die ursprüngliche.
2. Selbstreferenz: Aus allen hier vorgestellten Defintionen sticht dieser Aspekt klar heraus, er bildet sozusagen den Kern der Selbstorganisation. Durch Aktionen, die intern im System ausgelöst werden, ohne, dass von außen ein bestimmter Befehl dafür verantwortlich war, werden die Parameter des Systems geändert. Die Änderung der Parameter hat vermutlich wieder Auswirkungen auf eine neue, systeminterne Aktion, die ausgelöst werden könnte. Somit besteht eine Referenz zwischen den Zuständen eines Systems.
3. Redundanz: Es gibt in selbstorganisierenden Systemen keine Trennung zwischen organisierenden, gestaltenden und lenkenden Teilen. Alle Komponenten sind prinzipiell Gestalter, was zu Redundanz führt.
4. Autonomie: Systeme, die selbstorganisierend sind, sollten möglichst autonom in Bezug auf die Beziehungen und Interaktionen sein, die im System stattfinden, d.h., dass ein solches System keinen Anstoß von Außen braucht, um sich selbst in bestimmter Form zu verwalten. Natürlich befindet sich ein System immer in irgendeiner Umgebung, mit der es normalerweise auch Informationen austauscht. Dies verletzt auch nicht die Prämisse der Autonomie. [20]

3.2.3 Aufbau Der Weg, damit ein selbstorganisierendes System auch diesen Namen verdient, ist ziemlich lang und führt über verschiedene Zwischenstufen. Im folgenden Abschnitt wollen wir selbstorganisierende Systeme genauer charakterisieren. Dabei wird unser System S als Blackbox dargestellt, die folgende Ein- und Ausgangsparameter besitzt, was auch Abbildung 4 veranschaulicht:

- Reguläre Eingabevariable $r \in R_S$: Diese bezeichnet alle Eingaben ins System, die prinzipiell von jeder Art Sensor oder anderen Kommunikationsverbindungen stammen kann. Eingabevariablen können als Funktion bezüglich einer Zeit t aufgefasst werden.
- Ausgabevariable $o \in O_S$: Diese bezieht sich auf alle Werte und Attribute, die von S beeinflusst und der Umgebung wahrgenommen werden können. Ausgabe geschieht zu einer Zeit $t+1$, ist somit also auch wieder als Funktion bezüglich der Zeit aufzufassen.
- Kontrolleingabevariable $c \in C_S$: Solche Variablen beziehen sich auf eine andere Art der Eingabe, die das Verhalten des Systems beeinflusst. Sie sind auch wiederum zeitabängig.
- Eingabevariable $i \in I = R_S \cup C_S$
- Das Verhalten $\beta : I \mapsto O_S$ ist eine totale Relation, für die gilt: $\forall i \in I$ $\exists o \in O_S$ mit $\beta(i) = o$ [8, S.2] [11, S.10]

3.2.4 Stufen zur Selbstorganisation Mit Hilfe dieses vereinfachten Systemsmodells können wir nun die Konzepte voneinander abgrenzen, die zur Selbstorganisation führen und die Abhängigkeiten kurz erläutern. Auf Grund der sich

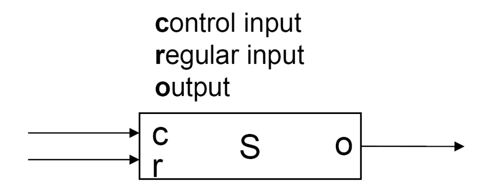


Abbildung 4. Modellierung eines einfachen Systems
[11, S.10]

sehr ähnelnden Ausdrücke im Englischen und fehlender deutscher Fachausdrücke wird im Weiteren dort, wo es als notwendig erscheint, der Originalausdruck verwendet:

- Adaptiv: Ein System S darf sich als „adaptiv“ bezüglich I bezeichnen, wenn $\forall(i \in I)$ gilt: $o = \beta(i) \Rightarrow p(i, o) \in W$. Dabei ist W eine Menge zeitabhängiger Funktionen, die ein Akzeptanzkriterium angibt, welches die Leistungsfunktion $p : (I \times O_S) \mapsto (T \mapsto R^n)$ einhalten muss.
- Manageable: Solche Systeme sind durch die geeignete Wahl eines $c \in C_S$ adaptiv.
- Self-Manageable bezeichnet ein bezüglich $I = R_S \cup C_S$ adaptives System. Außerdem kann dessen Kontrolleingabe c aus R_S und O_S berechnet werden kann
- Self-Managing sind Systeme, die adaptiv bezüglich R_S sind und für die $C_S = \emptyset$ gilt.

Man kann zeigen, dass sich jedes self-manageable System S in ein self-managing System S' verwandeln kann. Ein System ist nur selbstorganisierend, wenn es self-managing und adaptiv bezüglich seiner Struktur ist und nur auf dezentrale Steuerungsmechanismen zurückgreift. Dabei bedeutet „adaptiv bezüglich seiner Struktur“, dass es für die Adaption dynamisch seine Struktur ändern kann. Damit lässt sich folgende Teilmengenbeziehung feststellen, die auch in Abbildung 5 dargestellt wird: *selbstorganisierend* $\subseteq$ *self-managing* $\subseteq$ *self-manageable* $\subseteq$ *manageable* $\subseteq$ *adaptiv* [8, S.3-7] [11, S.11-14]
Um über die Struktur eines Systems sprechen zu können, müssen uns bestimmte Systemparameter bekannt sein. Der wohl wichtigste Punkt hierbei ist die Systemarchitektur, die wir in einem späteren Kapitel genauer betrachen werden.

3.2.5 Ameisenalgorithmus als Beispiel Als letztes sind wir noch ein Beispiel zur Selbstorganisation schuldig. Wir werden dazu kurz in die Biologie einsteigen und uns den Ameisen widmen, die sich in Kolonien organisieren, ohne eine zentrale Instanz zu haben, die das Vorgehen vorgibt. Wir nehmen den Fall an, dass wir zwei hungrige Ameisen haben A und B haben, die beide auf der Suche nach Futter sind. Von ihrem Ameisenhügel führen normalerweise mehrere Wege zur nächsten Futterquelle. Der Einfachheit halber wollen wir annehmen,

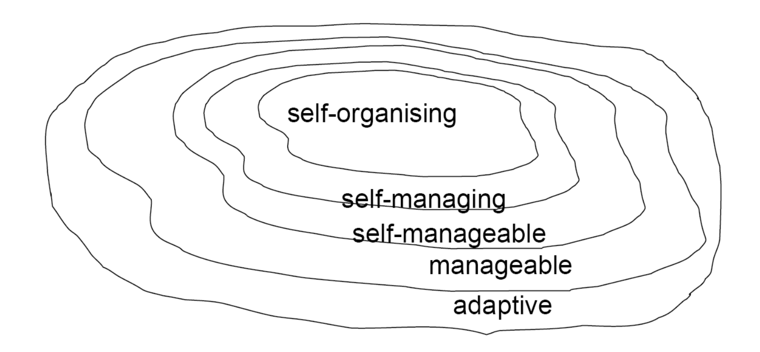


Abbildung 5. Die Abstufungen und Beziehungen der Begriffe [11, S.12]

dass in diesem Fall nur zwei relevante Wege dorthin führen, bezeichnet als s_1 und s_2, wobei wir annehmen, dass $|s_1| < |s_2|$. Ameise A macht sich nun über Weg s_1 auf die Reise zum Futter, B benutzt die Strecke s_2. Während dieser Reise hinterlassen die Ameisen Duftstoffe, sogenannte Pheromone auf dem Weg, die mit der Zeit an Stärke verlieren. Wenn nun eine Strecke besonders gut ist, wird auf ihr der Pheromonspiegel immer höher sein, als auf anderen, was die Ameisen wieder dazu verleitet, diese Strecke zu benutzen. Die anderen Strecken werden nur noch selten benutzt.
Dieser Ameisenalgorithmus, der im Prinzip so auch seinen Weg in Softwareimplementierungen gefunden hat, ist ein typisches Beispiel für Selbstorganisation: Die Ameisen befolgen immer dieselbe Regel, welche sie den Weg auswählen lässt, den andere Ameisen zuvor schon als den besten identifiziert haben. Eine einzelne Ameise könnte niemals systematisch mit vertretbarem Aufwand nach dem kürzesten Weg suchen. Nur eine ganze Kolonie ist ihrem seinem adaptiven Verhalten dazu in der Lage. Eine Problematik dieses Verfahren lässt sich hier allerdings auch schon erkennen: Ist die bevorzugte Route einmal gesperrt oder plötzlich nicht mehr vorhanden, benötigt das Verfahren wieder eine gewisse Anlaufzeit, um gute Ergebnisse zu liefern. Um diesen nicht tragischen, aber unbequemen Effekt abzumildern, könnte man den Zufall ins Spiel bringen, der einen gewissen, zu ermittelnden Prozentsatz von Ameisen auf neue, noch nicht untersuchte Pfade schickt, um diese zu erkunden. Somit könnte beim Ausfall der besten Route schneller ein guter Ersatz für diese gefunden werden. [11, S.5-7]

3.3 Emergenz

Im Kapitel zuvor haben wir gesehen, dass sich selbstorganisierende Systeme selbst verwalten können, indem sie auf Änderungen reagieren, ohne, dass diese Reaktion von einer zentralen Instanz vorgeschrieben wird. So schön diese Dezentralität auch sein mag, durch die erwähnte Komplexität selbstorganisierender Systeme kann es manchmal zu unvorhergesehen Effekten kommen, die man als emergentes Verhalten, oder kurz Emergenz bezeichnet.

3.3.1 Definition Der Begriff Emergenz leitet sich vom lateinischen „emergere“ ab, was soviel heißt wie „hervorkommen“ oder „sich zeigen“. Es tauchen also plötzlich Phänomene auf, die nur in großen Systemen mit vielen Komponenten auftreten können, aber nicht durch eine Komponente alleine auslösbar sind. Exakter ausgedrückt ist Emergenz „die spontane Herausbildung von Phänomenen oder Strukturen auf der Makroebene eines Systems auf der Grundlage des Zusammenspiels seiner Elemente. Dabei lassen sich die emergenten Eigenschaften des Systems nicht offensichtlich auf Eigenschaften der Elemente zurückführen, die diese isoliert aufweisen“. [19] Ein prägnanter Satz, der zum besseren Verständnis dieses Begriffs, führt lautet: „Das Ganze ist mehr als die Summe seiner Teile.“ [11, S.18] Dieser reflektiert auch wieder die Tatsache, dass enorme Komplexität und das Zusammenspiel vieler Einzelteile zu Emergenz führen. Man kann die positiven Effekte der Emergenz nutzen, wenn man sie richtig versteht. Gleichzeitig wird es häufig vorkommen, dass man die negative Seite von Emergenz zu spüren bekommt. Die Kunst ist es nun, die positiven Effekte zu nutzen und dabei negative Phänomene zu vermeiden, was wohl nicht immer gelingen wird. Emergenz wird in der Philosophie schon seit über einem Jahrhundert untersucht, wohingegen in der Informatik und den Ingenieurswissenschaften das Studium emergenten Verhaltens in technischen Systemen noch ein ziemlich junges Forschungsgebiet ist.
Der Begriff der Emergenz lässt sich noch feiner aufspalten. Wir unterscheiden zwischen starker und schwacher Emergenz. Beide werden nun kurz charakterisiert, immer mit der Einbettung in den philosophischen Hintergrund:

3.3.2 Schwache Emergenz Die schwache Emergenz basiert auf drei Thesen: der These des physikalischen Monismus, der These der systemischen Eigenschaften und der These des synchronen Determinismus. Die These des physikalischen Monismus besagt, dass alle Systeme, einschließlich solcher, die emergente Eigenschaften zeigen, nur aus physisch realen Komponenten bestehen. Es gibt keine übernatürlichen Kräfte, die von Außen irgendwie solches Verhalten hervorrufen können. Die These der systemischen Eigenschaften trifft das oben beschriebene Verhalten, das besagt, dass das System als Ganzes Eigenschaften zeigen kann, die einzelne Komponenten niemals alleine vorweisen können. Die These des synchronen Determinismus besagt zudem noch, dass Eigenschaften sowohl von der Mikrostruktur abhängen, die Abhängigkeit allerdings auch in umgekehrter Richtung vorhanden ist. Das bedeutet, dass die Änderung an einem von beiden gleichzeitig auch eine Veränderung des Anderen bewirkt. [11, S.20]

3.3.3 Starke Emergenz Die starke Emergenz beinhaltet alle Thesen der schwachen Emergenz, enthält aber zusätzlich die Forderung nach der Irreduzibilität. Dabei gilt eine systemische Eigenschaft als irreduzibel, wenn sie prinzipiell nicht aus dem Verhalten, den Eigenschaften und der Struktur von Komponenten hervorgeht. Dabei ist es gleichgültig, ob die Komponenten voneinander isoliert sind oder sich in einer anderen Konfiguration befinden. Eine reduzierte Erklärung existiert aber nicht und wird es auch nie geben. Man könnte also zusammenfas-

send sagen, dass starke Emergenz es unmöglich macht, die systemische Eigenschaft im Mikro- und im Makrokosmos zu analysieren. [11, S.22]

3.3.4 Entstehung Für unsere Zwecke werden wir im Weiteren nur die schwache Emergenz betrachten. Emergenz tritt in Organic Computing Systemen öfters auf als ein Phänomen, das sich zunächst nicht erklären lässt, wenn wir das System und seine Komponenten nicht ausreichend im Vorfeld spezifiziert haben. Im Folgenden betrachten wir nun, wie Emergenz beispielsweise entstehen kann:

- Bei Selbstorganisation kann, wie schon besprochen, Emergenz auftreten. Den besprochenen Ameisenalgorithmus kann man auch als positiv genutzte Emergenz ansehen. Er erfüllt alle Thesen der schwachen Emergenz.
- Emergenz kann auch auftreten, obwohl es in einer Implementierung keinen Algorithmus gibt, der dieses Verhalten direkt spezifiziert. Ein Beispiel dafür wäre ein Roboter, der Kisten aufeinander stapelt. Der Roboter weiß nichts davon, dass er dabei ist, einen großen Turm zu bauen. Er kennt nur die Aufgabe, jeweils eine Kiste an einen bestimmten Ort zu stellen und erfüllt diese mit höchster Präzision, so dass am Ende ein Turm als emergenter Effekt entsteht.
- In verteilten Systemen kann Emergenz auch positiv genutzt werden. Beispielsweise können Knoten in einem Netzwerk selbstständig Informationen untereinander austauschen und dadurch implizit neue Informationen über den Zustand des gesamten Netzes bekommen. Man spricht hier von Emergenz als Folge interaktiver Komplexität.
- Emergenz kann auch eine Folge des Unvorhersehbaren sein. Beispielsweise wird in einem realistischen Bevölkerungsmodell die Wachstumsrate nicht konstant sein, sondern von der aktuellen Populationsgröße abhängen.
- Emergenz kann auch im Sinne von nicht komprimierbarer Entwicklung auftreten. Dabei wird ein System als emergent bezeichnet, wenn der Gesamtzustand des Systems, der durch eine gewisse Mikrodynamik erreicht wurde, von der Mikrodynamik selber und externen Bedingungen abhängt und diese nur per Simulation abgeleitet werden können. [11, S.24 f.]

Zusammenfassend können wir sagen, dass Emergenz und Selbstorganisation sehr eng ineinander verzahnt sind. Selbstorganisierende Systeme müssen mit Emergenz auf Grund ihrer Komplexität leben. Sie kommt auf Grund des Zusammenspiels der vielen Elemente auf einer unteren Ebene zustande. Diese Emergenz kann positive Effekte haben, kann aber auch Nachteile mit sich bringen

3.4 Architekturtypen

Um den Anforderungen eines Organic Computing Systems gerecht zu werden, gibt es verschiedene Möglichkeiten, das System zu designen. Den Aufbau eines Systems, bzw. die Anordnung verschiedener Subsysteme und Komponenten, die zusammen ein großes System bilden, nennt man Architektur. In diesem Abschnitt werden kurz verschiedene Architekturtypen vorgestellt und am Schluss anhand eines konkreten Beispiels illustriert.

3.4.1 Observer/Controller Der beliebteste Architekturtyp nennt sich Observer/Controller. Diese Architektur wurde an der Universität Karlsruhe und der Universität Hannover entwickelt und entsteht durch geeignetes Zusammenfügen und Interaktion einzelner Bausteine, die im folgenden Schrittweise angeführt und erklärt werden.

- System under observation and control (SuOC): Dies ist unser eigentliches System, das zu überwachen und kontrollieren ist. Es zeichnet sich meist durch hohe Komplexität aus, die es erfordert, automatische Mechanismen zur Administration zu verwenden. Das System kann wiederum aus vielen interagierenden Elementen bestehen.
- Eingaben: Hierunter versteht man all das, was dem System zum Arbeiten übergeben wird. Dies können Parameter aus der Umgebung oder von anderen komplexen Systemen sein.
- Ausgabe: Alles, was das System nach Ausführung seiner Operationen wieder ausgibt, wird unter diesem Punkt zusammengefasst. Dies können verschiedenste Informationen seien, angefangen von Kontrollsignalen für andere Geräte, bis zum aktuellen Status des SuOC, der an andere angeschlossene Systeme oder die Umwelt ausgegeben wird.
- Observer: Ein Hauptbestandteil des Systems stellt diese Beobachtungseinheit dar. Sie ist dafür zuständig, das SuOC in irgendeiner Form zu überwachen. Dazu greift sie alle erhältlichen Informationen vom SuOC ab und gewinnt dadurch ein Bild vom aktuellen Systemstatus. Eine weitere, schwierigere Aufgabe des Observers ist es, aus den aktuellen Daten, die er durch Messungen bekommt, zukünftiges, auch emergentes Verhalten vorherzusagen. Dafür müssen aber die erhaltenen Informationen geeignet interpretiert werden, was wiederum eine Schwierigkeit bei einer konkreten Implementierung aufwirft.
- Observation model: Um zu wissen, welche Informationen aus dem System besonders wichtig sind, benötigt die Beobachtungseinheit ein sogenanntes „observation model“. Hier wird definiert, welche Informationen aus dem System beobachtet und ausgelesen werden sollen, in welcher Form sie analysiert und aggregiert werden und wie die Vorhersage des zukünftigen Systemstatus zustande kommt. Dieses Modell kann allerdings dynamisch verändert werden. Dazu sendet die Kontrolleinheit eine Rückmeldung (Feedback) an den Observer, über den ausgeführten Kontrollvorgang. Aus diesem Feedback können Rückschlüsse gezogen werden, welche Attribute und Eigenschaften im aktuellen Kontext interessant erscheinen und mit welcher Gewichtung sie zu berücksichtigen sind.
- Controller: Die Kontrolleinheit ist der zweite wichtige Pfeiler innerhalb einer Observer-Controller Architektur. Die vom Observer aufgesammelten und interpretierten Informationen über das SuOC werden an den Controller weitergereicht. Dieser hat nun die Aufgabe, das Verhalten des Systems in die richtige Richtung, die von der Umwelt dem System in Form von Zielen vorgegeben wird, zu steuern, beispielsweise mit Blick auf Emergenz. Es sollen also nur gewünschte emergente Effekte auftreten. Unerwünschte Verhaltensweisen sollen möglichst schnell und effizient unwirksam gemacht werden. Mit

Ziel, all dies einzuhalten, versetzt der Controller das System in einen neuen Zustand, der im besten Fall alle diese Kriterien erfüllt. [12, S.9 f.]

Wenn alle oben erwähten Bausteine in geeigneter Weise zusammenarbeiten, spricht man von einer Observer-Controller-Architektur. Die untenstehende Abbildung 6 illustriert dies.

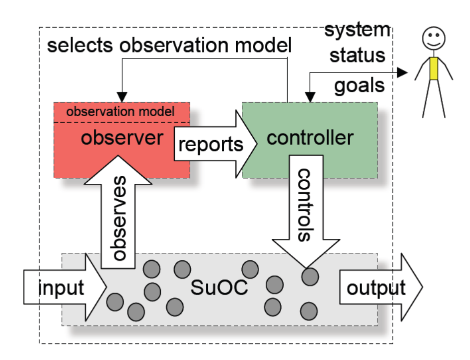


Abbildung 6. Observer-Controller
[12, S.11]

Darüber hinaus kann man diese Architekturform noch feiner untergliedern. Man kann hier hinsichtlich zweier Parameter unterscheiden:

- (De-)Zentral: Eine Architektur wird als zentral bezeichnet, wenn sie für das Gesamtsystem, das wiederum aus einzelnen Subsystemen bestehen kann, jeweils nur einen einzigen Observer und Controller zur Verfügung stellt. Dezentrale Architekturen sehen für jedes Subsystem eine eigene Beobachtungs- und Kontrolleinheit vor.
- Einstufig/Hierarchisch: Als hierarchisch bezeichnet man eine Architektur, die pro (Sub-)System mehrere Observer und Controller hat, die sich beispielsweise wieder gegenseitig steuern. Diese werden in Schichten angeordnet, d.h. ein Observer/Controller-Paar hängt wiederum in beiden Richtungen von je einem solchen Paar wieder ab. Diese Schichtenarchitektur ist auch grundlegender Baustein des ISO/OSI-Modells. Dies kann für komplexe Systeme nützlich sein, in denen jeder Observer und Controller wiederum nur eine einzelne oder wenige Aufgaben erfüllt und seine Ergebnisse an die anderen Schichten weiterreicht. Einstufige Systeme dagegen besitzen nur einen Observer und Controller für das ganze (Sub-)System.

Diese Merkmale lassen sich jetzt in den verschiedensten Varianten miteinander kombinieren. Dies ist auch in Abbildung 7 dargestellt.

Neben den klassischen Kombinationsmöglichkeiten gibt es auch noch ungewöhnlichere Kombinationen, die keine Eigenschaft in ihrer Reinform, wie oben

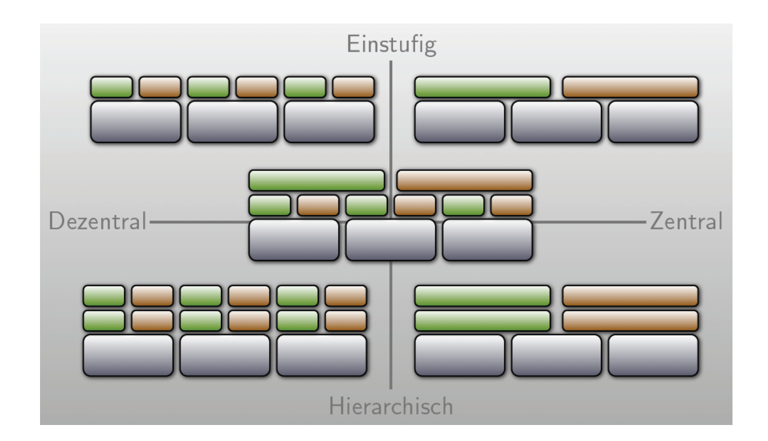


Abbildung 7. Verschiedene Architekturformen
[15]

beschrieben darstellen. Diese Architekturformen des Observer-Controller Modells nennt man „hybrid". [15]

3.4.2 Autonomous Middleware for Ubiquitous eNvironements Ein Vertreter einer auch in der Praxis implementierten Architektur ist die AMUN Architektur. AMUN steht dabei für „Autonomus Middleware for Ubiquitous eNvironments". Dabei ist das Ziel einer „ubiquitous environment", dass die Umgebung den Menschen durch Allgegenwärtigkeit von (kleinen) Rechnern unterstützt. AMUN erweitert im Wesentlichen das uns schon bekannte Observer-Controller Modell und verfeinert dies. Hier soll nur ein kurzer Überblick gewährt werden. [14, S.2] [22] AMUN ist ein Ansatz einer autonomen/organic Middleware für eine Umgebung innerhalb eines Gebäudes. Eingesetzt wird sie beispielsweise im Rahmen des Smart Doorplate Projektes an der Universität Augsburg. Ziel dieses Projektes war die Umsetzung eines intelligenten Leitsystems, das innerhalb eines Gebäudes Besucher dynamisch den Weg zum richtigen Büro oder Raum weist. Die Middleware AMUN selbst besteht aus vier Hauptteilen, wie auch in Abbildung 8 veranschaulicht wird:

- Transport Interface: Diese Schnittstelle hat als Aufgabe, die Nachrichtenübermittlung vom benutzten Transportsystem zu entkoppeln. Somit wird eine gewisse Abstraktion geschaffen. In der unten stehenden Abbildung wurde für die Transportfunktionalität ein p2p Ansatz gewählt.
- Event Dispatcher: Dieser verteilt die ein- und ausgehenden Nachrichten korrekt an die jeweiligen Instanzen.
- Service Interface: Dieses dient als Schnittstelle zwischen AMUN und einem angebotenen Service. Ein Dienst, der AMUN nutzen will, muss dieses Service Interface implementieren, damit es auch Nachrichten empfangen kann, die vom Event Dispatcher bereitgestellt wurden.

- Autonomic Manager: Unter diesem Schlagwort versteckt sich die Hauptkontrollinstanz eines AMUN Knotens. Hier werden alle relevanten Informationen der lokalen Dienste und des Monitors gesammelt. Dieser Manager besteht wiederum aus mehreren einzelnen interagierenden Komponenten. Der Konfigurator ist verantwortlich für die Konfiguration der Dienste auf einem Knoten, der Kontrollalgorithmus wertet den aktuellen Status mit Hilfe einer Metrik aus, die aus dem Metrik Pool stammt. Er ist für die Rekonfiguration von Diensten zuständig. Service und Monitor Info Space sind die Speicherplätze für den aktuellen Service bzw. Monitor Status.
- System Monitor: Er überwacht das System nach definierten Gesichtspunkten und sammelt Informationen darüber, wie zum Beispiel über den aktuell vorhanden freien Speicher und die Prozessorauslastung. Diese Informationen werden für die Berechnungen benötigt, die von den anhand der Metriken durchgeführt werden. [12, S.46]

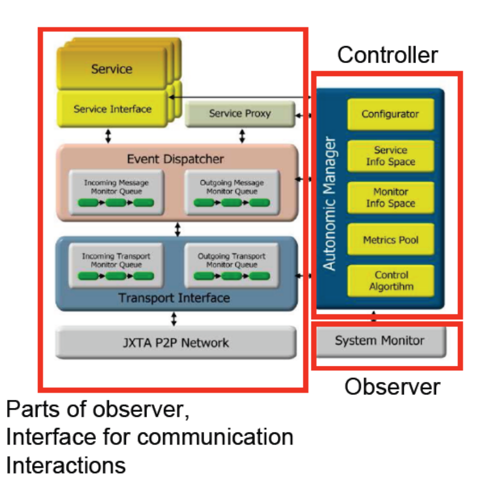


Abbildung 8. Architektur der AMUN Middleware
[12, S.47]

Wenn man die gesamte AMUN Architektur mit der Observer-Controller Architektur vergleicht, stellt man fest, dass der System Monitor genau unser Observer ist und der Autonomic Manager der Controller. [12, S.47]

3.4.3 Zeroth Level Classifier System (ZCS) Ein weiteres praktisches Beispiel für eine verfeinerte Observer-Controller Architektur ist das von Stewart W. Wilson entwickelte Zeroth Level Classifier System (ZCS). Es besteht aus mehreren Komponenten, was auch in Abbildung 9 demonstriert wird:

- Sensoren/Detektoren: Sie empfangen und messen Parameter aus der Umwelt und geben sie binär an die Classifier [P] weiter.

- Classifier [P]: Hierbei handelt es sich im wesentlichen um Tupel aus Bedingung, Aktion und Stärke. Bedingung und Aktion sind binär codiert, die Stärke kann dezimal angegeben werden. # werden als Wildcards angesehen und können für 0 oder 1 stehen. Die Classifier in [P] werden duch einen gezielten genetischen Algorithmus erzeugt. Die genaue Funktionsweise dieses Algorithmus kann je nach konkreter Implementierung variieren und soll hier der Einfachheit halber vernachlässigt werden. Im Prinzip sorgt er aber dafür, dass alte Classifier, die sich als zu schwach herausgestellt haben, verworfen und neue, bessere erstellt werden. Ebenso verhält es sich mit dem sogenannten Covering, das auch nur zur Auffüllung von [P] verwendet wird, wenn [M] zum Beispiel leer wäre. Covering erstellt dann einen neuen Classifier, dessen Bedingung zur eintreffenden passt und fügt ihn in [P] und [M] ein. Die Stärke des neuen Classifiers wird anhand der Durchschnittsstärke aller bisher vorhandenen Classifier ermittelt, die Aktion, die er ausführt, wird zufällig bestimt. Somit kann [M] nie leer sein und der Algorithmus läuft planmäßig weiter.
- Match Set [M]: Im Match Set landen die Classifier aus [P], welche dieselbe Bedingung aufweisen, die auch vom Sensor an [P] weitergegeben wurde. Dabei werden natürlich auch die Wildcards berücksichtigt. Somit wird also aus einer großen Anzahl an Classifiern eine kleinere ausgewählt, die eine zur vorgegebenen Bedingung passende Aktion ausführen können.
- Action Set [A]: Für das Action Set wird aus [M] ein Classifier nach der Roulette-Wheel-Methode ausgewählt und dorthin kopiert. Bei der Roulette-Methode dient die Stärke der jeweiligen Classifier als Gewichtung für die Auswahl. Haben wir beispielsweise in [M] drei Classifier A, B und C. A besitzt Stärke 50, B Stärke 30 und C Stärke 20. Als nächstes wird eine Zufallszahl i zwischen 1 und 100 ausgewählt. Für $1 \leq i \leq 50$ wird A ins Action Set kopiert, für $51 \leq i \leq 80$ B und für $81 \leq i \leq 100$ C. Zudem werden alle anderen Classifier in [M], welche dieselbe Aktion beinhalten, nach [A] verschoben. An die Aktuatoren und Effektoren wird nun die ausgewählte Aktion weitergeleitet, welche sie wiederum an die Umwelt weitergeben. Abschließend findet eine Bewertung der Classifier statt. Classifier, die in [A] enthalten sind, werden mit Hilfe einer Belohnung, die aus der Umwelt kommt, aufgewertet, d.h. ihre Stärke wird erhöht und somit ist es wahrscheinlicher, dass sie beim nächsten Durchlauf bei passender Bedingung für das Match Set ausgewählt werden, die anderen Classifier, die nur in [M] und nicht in [A] sind, werden entsprechend abgewertet. [23, S.3-8] [15]

Durch das Zusammenspiel der Umwelt und der verschiedenen Komponenten dieses Systems, ist es offensichtlich möglich, dass sich das System in gewissem Rahmen selbst steuern kann. Durch den Belohnungsmechanismus ist es an die Umwelt gekoppelt und bekommt wieder ein Feedback über die Qualität der zuvor ausgewählten Aktion, wie das auch im Observer-Controller Schema ist. Das System ist also bemüht, eigenständig, ohne Eingriff des Menschen, immer bessere Lösungen für dasselbe Problem zu finden.

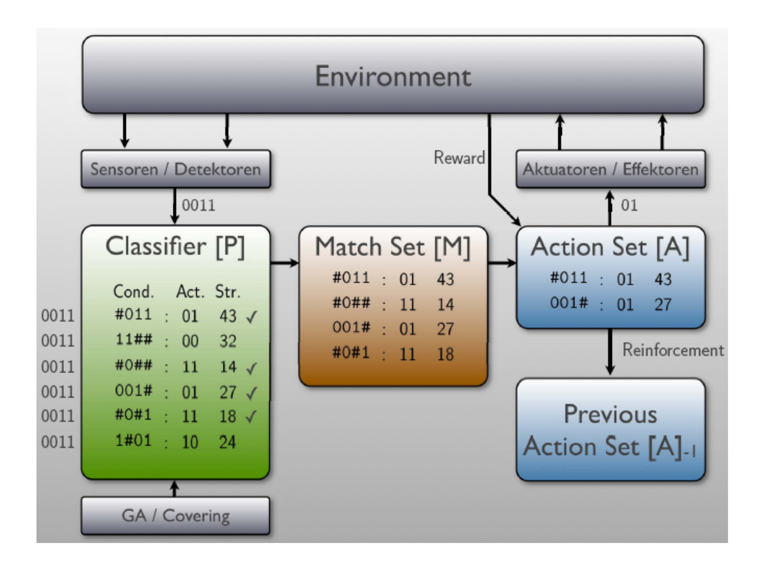


Abbildung 9. Zeroth Classifier System (ZCS)
[15]

4 Zusammenfassung und Ausblick

Organic Computing ist sicherlich ein aufstrebender Zweig innerhalb in der Informatik. Da unsere Welt und damit auch unsere Systeme, die größtenteils schon diese Welt steuern, zumindest aber die Steuerung dieser untersützen, immer komplexer werden, wird zwangsläufig das Interesse an Beherrschung dieser Komplexität stärker werden, wofür Organic Computing einen möglichen Ansatzpunkt bietet. Dazu muss aber noch viel Grundlagenforschung betrieben werden, um Prinzipien des Organic Computing in alltäglichen Systemen einsetzen zu können. Das Phänomen der Selbstorganisation muss dazu noch genauer erforscht werden. Ebenso muss man sich auch noch Gedanken über die konkrete und gewinnbringende Nutzung solcher Phänomene in Systemen machen. Am Ende dieser zwei Phasen steht dann der schwierigste Teil noch bevor, die konkrete Implementierung. [10, S.334 f.] Das Smart-Doorplate-Projekt hat gezeigt, dass es möglich ist, solche Systeme zu entwickeln. In den nächsten Jahren werden wohl noch komplexere Systeme dieser Art realisiert und marktreif werden. Wohin uns die Zukunft genau führen wird, ist natürlich, wie in der Informatik so oft, nicht genau vorherzusehen.

Literatur

[1] C. Bobda and R. Wanka. Organic Computing - Introduction, Motivations, Overview, 2006. http://www12.informatik.uni-erlangen.de/edu/OC/SS06/1_OC_introduction.pdf, abgerufen am 24.02.2008.

[2] IBM. An architectural blueprint for autonomic computing. http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf, abgerufen am 24.02.2008.
[3] IBM. Autonomic Computing - Adoption Model. http://www-03.ibm.com/autonomic/about_get_model.html, abgerufen am 24.02.2008.
[4] IBM. Autonomic Computing - The 8 Elements. http://www.research.ibm.com/autonomic/overview/elements.html, abgerufen am 24.02.2008.
[5] IBM. Autonomic Computing - The Solution. http://www.research.ibm.com/autonomic/overview/solution.html, abgerufen am 24.02.2008.
[6] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing, January 2003. http://www.research.ibm.com/autonomic/research/papers/AC_Vision_Computer_Jan_2003.pdf, abgerufen am 24.02.2008.
[7] C. Lucas. Self-Organizing Systems (SOS) FAQ. http://www.calresco.org/sos/sosfaq.htm#1.2, abgerufen am 24.02.2008.
[8] G. Mühl, M. Werner, M. A. Jaeger, K. Herrmann, and H. Parzyjegla. On the Definition of Self-Managing and Self-Organizing Systems. http://www.kbs.cs.tu-berlin.de/publications/fulltext/kivs2007_2.pdf, abgerufen am 24.02.2008.
[9] C. Müller-Schloer, H. Schmeck, and T. Ungerer. Antrag auf Einrichtung eines neuen DFG Schwerpunktprogramms, Februar 2004. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/download/SPP-AntragOC20040216.pdf, abgerufen am 24.02.2008.
[10] C. Müller-Schloer, C. von der Malsburg, and R. P. Würz. Organic Computing. *Informatik Spektrum*, 27(4):332–336, August 2004.
[11] S. Mostaghim and H. Schmeck. Organic Computing - Self-Organisation and Emergence. http://www.aifb.uni-karlsruhe.de/Lehre/Sommer2007/OC/unterlagen/OC2.a.pdf, abgerufen am 24.02.2008.
[12] S. Mostaghim and H. Schmeck. Organic Computing - Architecture of OC, April 2007. http://www.aifb.uni-karlsruhe.de/Lehre/Sommer2007/OC/unterlagen/OC3.pdf, abgerufen am 24.02.2008.
[13] M. Naylor. Glossary: Self-organisation. http://www.geos.ed.ac.uk/homes/mnaylor/Glossary.html, abgerufen am 24.02.2008.
[14] W. Trumler, J. Petzold, F. Bagci, and T. Ungerer. AMUN - An Autonomic Middleware for the Smart Doorplate Project. http://ubisys.cs.uiuc.edu/proceedings_04/amun.pdf, abgerufen am 24.02.2008.
[15] W. Trummler. Organic Computing, Vorlesung an der Universität Augsburg, April 2007.
[16] T. Ungerer. Smart Doorplate. http://www.informatik.uni-augsburg.de/lehrstuehle/sik/forschung/ubicomp/smartdoorplate/, abgerufen am 24.02.2008.
[17] Wikipedia. Architektur (Informatik). http://de.wikipedia.org/wiki/Architektur_%28Informatik%29, abgerufen am 24.02.2008.
[18] Wikipedia. Darwinismus. http://de.wikipedia.org/wiki/Darwinismus, abgerufen am 24.02.2008.
[19] Wikipedia. Emergenz. http://de.wikipedia.org/wiki/Emergenz, abgerufen am 24.02.2008.
[20] Wikipedia. Selbstorganisation. http://de.wikipedia.org/wiki/Selbstorganisation, abgerufen am 24.02.2008.
[21] Wikipedia. Self-management (computer science). http://en.wikipedia.org/wiki/Self-management_%28computer_science%29, abgerufen am 24.02.2008.
[22] Wikipedia. Ubiquitous Computing. http://de.wikipedia.org/wiki/Ubiquitous_Computing, abgerufen am 24.02.2008.

[23] S. W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.

Rules im Organic Computing

Bekr Turki Hatam

Universität Augsburg
bekralsultan@hotmail.com

Zusammenfassung Das Ziel des Organic Computing ist es, die in der Natur beobachteten Prinzipien technisch zu nutzen. Organische Computersysteme bestehen aus einer großen Anzahl von Einzelkomponenten und sie sollen die Fähigkeit zur Selbstorganisation haben. Aus dem Zusammenwirken von diesen Komponenten entstehen neue Strukturen bzw. Eigenschaften, die Emergenz bezeichnet werden. Emergenz ist ein wichtiger Aspekt der Selbstorganisation und zeigt wie die Beobachtung das lokale Handeln der Individuen eines sich selbstorganisierenden Systems ein globales beobachtbares Verhalten ergeben kann. In dieser Arbeit werden die Begriffen Emergenz und Selbstorganisation näher betrachtet. Unter Selbstorganisierenden Systeme versteht man auch Regelbasierte Systeme, und wie man durch Menge von Regeln und Wissen neues Wissen gewinnen kann. Die Notation der einzelnen Regel erfolgt im if then-Stil. Das Schließen von neuem Wissen geschieht nach zwei Strategien: Vorwärtsverkettung und Rückwärtsverkettung. Jedes regelbasierte System sollte einen Regelinterpreter haben, der entscheidet, welche Regeln angewendet werden, und steuert das regelbasierte System. Regelinterpreter in regelbasierten Systemen ist auf die Lösung bestimmtes Problem spezialisiert. Aber in der Praxis ändern die Unternehmen ihre Rahmenbedingung immer schnell, deswegen war es in letzten Jahren nötigt, neue Regelinterpreter zu entwickeln, die universell verwendbar sind. Solche Interpreter wurden Rule Engines genannt. Wie Rule Engines arbeiten und woraus sie bestehen und wie sie in der Praxis eingesetzt werden, wird auch in dieser Arbeit erläutert. Zum Schluss werden zwei Regelsprachen (RuleML und Drools) betrachtet und durch Beispiele ihre Unterschiede und Gemeinsamkeiten kennen gelernt.

1 Einführung

Die Zahl von technischen Systemen und deren Komplexität nimmt stark zu. Dies gilt sowohl für einzelne technische Geräte, von denen wir im Alltag umgeben sind, als auch für die Systeme, die die Infrastruktur für eine massive Vernetzung bilden. Das Verständnis der zunehmenden Komplexität gilt als eine der größten Herausforderungen der mittelfristigen Zukunft in den Techniken der Informationsverarbeitung. Sie will und muss beherrscht werden, will man weiterhin der rasanten Weiterentwicklung folgen können. Ansatzpunkt zum Verständnis der Komplexität sind organische Computersysteme inspiriert von Strukturen aus der Natur, wie z. B. das menschliche Nervensystem oder das selbst-organisierende

Verhalten von Ameisen bzw. Tierschwärmen. Bei *Organic Computing* handelt es sich um dynamische, sich den jeweiligen Umgebungsbedürfnissen von selbst anpassende Computersysteme. Diese Fähigkeit zur Selbstorganisation äußert sich bei einem Organischen Computersystem dadurch, dass er selbst-optimierend, selbst-konfigurierend, selbst-heilend, selbst-schützend und selbst-bewusst arbeitet [16].

Organische Computersysteme bauen auf einem Zusammenspiel von einer großen Anzahl von Einzelkomponenten auf. Dabei ist es wichtig, dass sich diese Einzelkomponente an die jeweiligen Anforderungen von außen anpassen und nicht starr den programmierten Vorgaben folgen [12]. Durch das selbstorganisierende und damit ohne zentrale Kontrolle stattfindende Zusammenwirken dieser oft großen Anzahl von Einzelkomponenten entstehen neue Strukturen bzw. Eigenschaften, die als Emergenz[1] bezeichnet werden.

Als selbstorganisierendes System nehmen wir ein intelligentes (eingebettetes) System wie Smart- Car, House, Office, Factory an. Hier sollte das System selbständig sein und die Notwendigkeit und Zuverlässigkeit von Veränderungen und ihren Ablauf überwachen. Dabei können im ersten Schritt einfache Regeln für den Selbstschutz angewandt werden. Z.B. bei „Smart Car“ soll das Auto sich an verschiedene Fahrer, Straßenverhältnisse anpassen. Das Auto soll auch in der Lage sein, die aktuellen besten Fahrstrecken zu berechnen und in speziellen Ereignissen oder Fällen mit anderen Fahrzeugen zu kommunizieren ohne die Aufmerksamkeit des Fahrers zu stören. Das Auto soll die Interaktion mit seiner Umgebung übernehmen und die Gräte des Fahrers in sein Netzwerk integrieren.

Die Einhaltung solcher Regeln setzt voraus, dass eine Selbstbeobachtung des eingebetteten Computers möglich ist, also die Erfassung des eigenen Zustands sowie des Zustands der Umgebung (des Kontexts). Die hierzu benötigte Systemfunktion wird als Observer bezeichnet. Zum Zweiten muss eine Selbstkontrolle möglich sein, um die Einhaltung der Regeln zu gewährleisten. Die dazu benötigte Controller-Funktion ist kompliziert, denn der organische Computer muss die Konsequenzen einer Veränderung durch Updates, durch veränderte oder neue Anwendungen abschätzen können, bevor die Änderung durchgeführt wird. Durch diesen Prozess wird am Ende für das System ohne Vorwissen eine Regelbasis gewonnen. Die Notation der einzelnen Regeln erfolgt im if then-Stil, d. h. wenn die Bedingung bei if wahr ist, wird die bei then angegebene Aktion ausgeführt.

Wie wir gesehen haben, spielen die Regeln in Organic Computing eine große Rolle. In dieser Arbeit wird zuerst den Begriff Emergenz und die Rollen von dem lokalen und globalen Verhalten bei selbstorganisierendem System erklärt. Im Kapital 3 wird den Begriff „Regel“ und wie werden die Regelbasierte Systeme aufgebaut erläutert. Natürlich sollten die gewonnenen Regeln verarbeitet werden, um die benötigten Informationen abzuleiten. Wie die Regeln interpretiert werden und was man mit Rule Engines meint, ist der Inhalt vom Kapital 4. Eine detaillierte Beschreibung von zwei verschiedenen Regelsprachen wird im Kapital 5 ausgeführt.

[1] Emergenz ist generell als ein Prozess verstanden, Vergleich Abschnitt2.

2 Lokales und globales Verhalten

Selbstorganisierende Systeme bilden eine geschlossene, geordnete Organisation um eine Funktion zu erfüllen. Daraus resultieren emergente Eigenschaften. Emergenz ist generell als ein Prozess verstanden, welcher zur Entstehung einer Struktur führt, die nicht direkt von den existierenden Bedingungen und den momentanen Kräften, die eine System kontrollieren, beschrieben werden kann [14]. Emergenz und selbstorganisierendes System sind sehr Nah verbunden und Emergenz ist ein wichtiger Aspekt der Selbstorganisation und zeigt wie die Beobachtung das lokale Handeln der Individuen eines sich selbst organisierenden Systems ein globales beobachtbares Verhalten ergeben kann. Von Emergenz redet man immer dann, wenn ein System die folgenden Eigenschaften hat (gemäß der Definition in der Zeitschrift „Emergence“ [11]):

- Das globale Verhalten ist von anderer Art als das der einzelnen Komponenten (insbesondere keine lineare Kombination der Einzelhandlungen).

- Die Entfernung einzelner Komponenten führt nicht zu einem Ausfall des gesamten Systems.

- Das globale Verhalten ist völlig neuartig im Vergleich mit den vorhandenen Komponenten, d.h. das emergente Verhalten scheint unvorhersehbar und nicht ableitbar von den einzelnen Komponenten des Systems zu sein, und es kann auch nicht auf diese reduziert werden.

Um den Unterschied zwischen Emergenz und Selbstorganisation abzugrenzen und vorzustellen, sehen wir die Abbildung 1. Selbstorganisation fokussiert sich auf die Systemumgebungsgrenzen, wenn die Grenze zwischen inneren und äußeren als Teilen betrachtet werden. Im Gegenteil dazu nimmt der Prozess des Emergenz Plätze an der Grenze zwischen dem System und seinen Bestandteilen, wenn die Grenzen zwischen dem lokalen und globalen, (Mikro und Makro) durch Individuen und gesamtes Verhalten gekreuzt werden, sehe Abbildung 1 [11]. Es unterscheidet zwischen lokalen (Low-Level) Komponenten und globalen (High-Level) Patterns.

Organic Computing besteht aus einer großen Anzahl von Einzel-Komponenten. Dabei wird von den Komponenten erwartet, dass sie sich nicht starr an einprogrammierte Vorgaben halten, sondern adaptiv auf Anforderungen von außen reagieren. Das Gesamt-Systemverhalten ergibt sich dann wieder aus dem Zusammenspiel der Elemente (Emergenz). Von zentraler Bedeutung für den technischen Einsatz der Prinzipien des Organic Computing wird daher ein tieferes Verständnis organischer Systemarchitekturen sein. Es geht dabei sowohl um ein Verständnis natürlicher emergenter Systeme wie auch um die Frage der Beherrschbarkeit von Selbstorganisation und Emergenz in technischen Systemen [12]. Beherrschung der Selbstorganisation in technischen Systemen bedeutet die Vision von freundlichen Systemen und wie das System seinen eigenen „Willen“ durchsetzt in dem, dass Systemen den Menschen bedient und nicht umgekehrt.

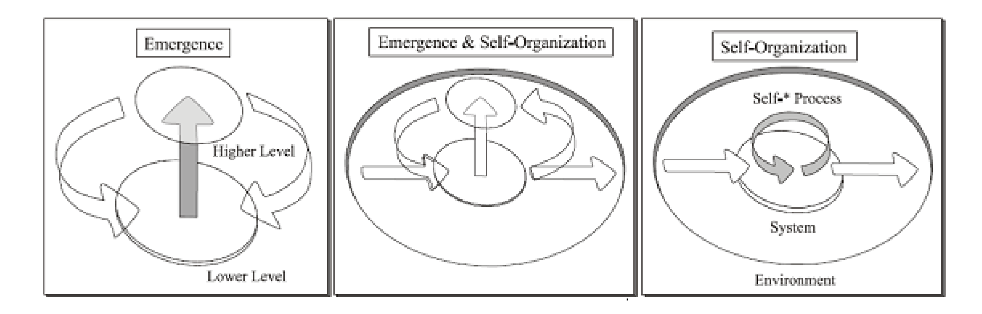


Abbildung 1. Emergenz und selbstorganisierendes System [11]

Bereits jetzt ist die Komplexität der uns umgebenden vernetzten informationsverarbeitenden Systeme zu groß, um noch eine effektive globale Steuerung vornehmen zu können.

Lokalität spielt bei der Beherrschung der Komplexität eine wesentliche Rolle d.h. aber nicht, dass ein globales Verhalten selbstorganisierender Systeme die Vereinigung des lokalen Verhaltens der einzelnen Systemkomponenten ist. Die einzelnen Systemkomponenten haben selbst kein Wissen über die gesamte Systemkomplexität, da sie auf Basis von Informationen agieren, welche an ihrem momentanen Standort verfügbar sind. Global notwendige Informationen werden anhand von Verteilungs- und Propagationsmechanismen lokal verfügbar gemacht. Deswegen erzeugen die Interaktionen zwischen den Systemkomponenten vielmehr ein größeres System als es alle einzelnen Komponenten könnten. Die Stabilität beim Zusammenspiel großer Zahlen von lokalen Steuerungen soll jedoch beobachtet werden.

3 Regelbasierte Systeme

Ein Regelbasiertes System ist ein Wissensbasiertes System[2] in dem regelbasiertes Schließen stattfindet [18]. Diese Technik entstand Mitte des zwanzigsten Jahrhunderts. Eine der Schlüsselaufgaben ist die Wissens-Aufbereitung, die sich mit der Verarbeitung von Informationen beschäftigt. In der Praxis sind Regelbasierte Systeme inzwischen weit verbreitet und in vielen Bereichen eingesetzt werden wie z.B. in der Medizin, in der Industrie bis hin zur Entwicklung von Computerspielen. Regelbasierte Systeme wurden lange Zeit als Prototypen für Expertensysteme[3] angesehen. In Abbildung 2 sieht man die verschiedenen Unterkategorien der wissensbasierten Systeme [7], auf die hier nicht näher eingegangen wird.

[2] Ein wissensbasiertes System (WBS) ist ein intelligentes Informationssystem, in dem Wissen mit Methoden der Wissensrepräsentation und Wissensmodellierung abgebildet nutzbar gemacht wird [17].

[3] Als Expertensystem (XPS) wird eine Klasse von Software-Systemen bezeichnet, die auf der Basis von Expertenwissen zur Lösung oder Bewertung bestimmter Problem-

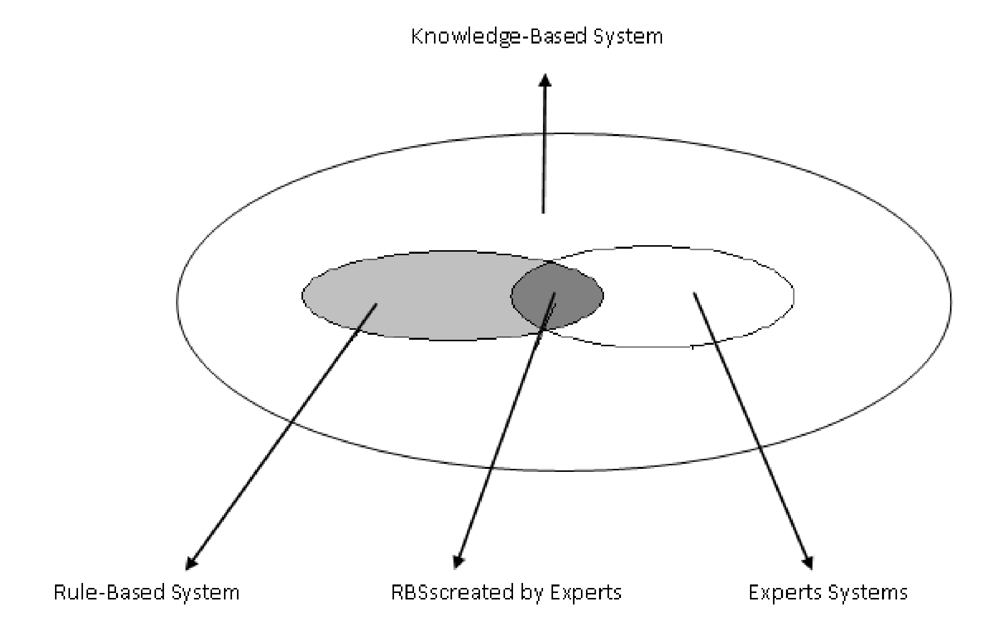


Abbildung 2. Verschiedene Arten von Wissensbasierten Systemen [7]

3.1 Was Sind die Regeln

Regeln sind formalisierte Konditionalsätze der Form [18]

$$Wenn(if) A dann(then) B \tag{1}$$

Mit der Bedeutung: Wenn A wahr (erfüllt, bewiesen) ist, dann schließe, dass auch B wahr ist.

wobei A und B Aussagen sind. Die Formel im „Wenn"- Teil einer Regel (also A in Eq. 1) wird als Prämisse, Bedingung oder Antezedenz der Regel bezeichnet (wird auch LHS, Left Hand Side, genannt), während die Formel im „Dann" - Teil (also B in Eq. 1) Konklusion oder Konsequenz genannt wird (wird auch RHS, Reight Hand Side, genannt). Wenn Prämisse einer Regel erfüllt ist, die Regel also angewendet werden kann, so sagt man auch, die Regel feuert. Gilt die Regelin Eq. 1 immer, also ohne Ausnahme, so spricht man auch von einer deterministischen Regel. Als Beispiel für eine Regel: Wenn ich müde bin, dann gehe ich ins Bett.

Die zwei Teile einer Regel können je nach Kontext unterschiedlich interpretiert werden; z.B.:

- Schlussregeln: wenn Prämisse, dann Konklusion.

stellungen dient. Beispiele sind Systeme zur Unterstützung medizinischer Diagnosen oder zur Analyse wissenschaftlicher Daten. [19].

– Hypothesenbildung (etwa bei Diagnose): wenn Evidenz, dann Hypothese.

– Produktionen: wenn Bedingung, dann Aktion.

Regeln wurden und werden gerne in Produktionssystemen zur Steuerung eingesetzt, wobei die Konsequenz einer Regel oft mit Aktion verbunden ist [3]. Eine einzelne Regel stellt (im Idealfall) eine Wissenseinheit dar aber eine Menge von Regeln zusammen mit einer Ausführungsstrategie stellt eine Art von Programm dar für die Lösung eines Problems oder einer Problemklasse - vgl. Prolog-Programm (Regelbasierte Systeme).

3.2 Aufbau eines Regelbasierten Systems

In diesem Abschnitt werden die Bestandteile eines Regelbasierten Systems kurz geklärt. Mehr darüber wird unter Rule Engine gegeben, sehe Abschnitt 3.5. Ein Regelbasiertes System besteht aus drei unterschiedlichen Bestandteilen:

1. *Datenbank / Arbeitsspeicher (Working Memory:* Der Arbeitsspeicher enthält eine Menge von Fakten[4] und Tätigkeiten, die den momentanen Zustand des modellierten Weltausschnitts beschreiben.
2. *Regelbasis (Rulebase):* Enthält die Regeln des Systems.
3. *Regelinterpreter:* Interpreter entscheidet, welche Regeln angewendet werden, und steuert das regelbasierte System.

3.3 Schließen von neuem Wissen (Inferenz)

Das Schließen von neuem Wissen aus vorhandenem Wissen und gegebenen Regeln wird Inferenz genannt. Im einfachsten Fall kann bei Eintreffen einer Bedingung schon aus einer Regel eine neue Information gewonnen werden. Es ist aber auch möglich, aus der Verkettung mehrerer Regeln Informationen zu gewinnen. Dabei werden zwei Strategien unterschieden, die Vorwärtsverkettung (Forward chaining) bzw. datengetriebene Inferenz (data-driven inference) und die Rückwärtsverkettung (backward chaining) bzw. zielorientierte Inferenz (goal-oriented inference)[4].

Beispiel 1: Sei die Wissensbasis enthält die Objekte: A,B,C,D,E,F,G,H,I,J,K,L,M, jeweils mit den Werten true, false, und die folgenden Regeln:

Abbildung 3 zeigt das zugehörige Regelnetzwerk, das eine graphische Präsentation der Regeln aus Beispiel 1 darstellt [4].

3.3.1 Vorwärtsverkettung

Bei der Vorwärtsverkettung wird versucht, die aus Regeln gewonnenen, neuen Informationen wiederum als Eingabewerte für neue Regeln zu verwenden, um so nochmals neue Informationen zu gewinnen. Wir betrachten nun diese zwei Regeln:

[4] Fakten stellen die Datengrundlage für die Rule Engine dar, vgl. Abschnitt 3.5.1.

$R1$:	**if**	$A \wedge B$	**then** H
$R2$:	**if**	$C \vee D$	**then** I
$R3$:	**if**	$E \wedge F \wedge G$	**then** J
$R4$:	**if**	$H \vee I$	**then** K
$R5$:	**if**	$I \wedge J$	**then** L
$R6$:	**if**	$K \wedge L$	**then** M

1. Wenn ich müde bin, dann gehe ich ins Bett!
2. Wenn ich ins Bett, Ziehe ich mir einen Schlafanzug an!

Der Zweite Regel ist von erster Regel abhängig, da die Bedingung zweiter Regel wahr wird, sobald der Sprecher ins Bett geht. D.h. es werden aus einer Information (in diesem Fall: der Sprecher ist müde) zwei neue Informationen (in diesem Fall: der Sprecher geht ins Bett und zieht ihm einen Schlafanzug an) aus zwei verschiedenen Regeln gewonnen. Aus den Konsequenzen von Regeln können also Informationen gewonnen werden, die wiederum zur Prüfung der Bedingungen anderer Regeln verwendet werden können. Der Ablauf der Vorwärtsverkettung setzt sich wie folgt zusammen, (sehe Anhang A, Abbildung 7):

1. *Matching:* Berechne alle Regeln, deren Bedingungen bei den aktuellen Variablenwerten erfüllt sind
2. *Konfliktlösung:* Trittkraft, wenn mehre Regel anwendbar sind (mögliche Lösung: Wähle die Regel, in welcher der Bedingungsteil sich auf möglichst neue Eintrage in der Datenbasis bezieht
3. *Durchführung:* Führe die Anweisungen der ausgewählten Regel aus.

Ein entsprechender Algorithmus kann wie in Anhang A, Abbildung 8 angegeben formuliert werden

3.3.2 Rückwärtsverkettung Die Rückwärtsverkettung geht von einem Zielpunkt aus und versucht dessen Bedingungen zu bestätigen, (sehe Abbildung A, Abbildung 9). Merkmale der Rückwärtsverkettung: Man geht vom Zielpunkt aus, stellt Hypothesen auf, überprüft die Vermutungen. Wir betrachten wieder die Regel im obigen Abschnitt und wenden dieses Verfahren an. Die folgenden Schritte würden rückwärts durchgeführt werden:

1. Ich ziehe dann einen Schlafanzug an, wenn ich ins Bett gehe-
2. Und ich gehe dann ins Bett, wenn ich müde werde!.

Also hängt der Wahrheitsgehalt der These, dass der Sprecher einen Schlafanzug anzieht, von zwei Bedingungen ab. Dass diese beiden Bedingungen darüber hinaus direkt voneinander abhängen, ist hier nur Zufall. Es könnten auch beliebig viele voneinander unabhängige Bedingungen sein, (Das allgemeine Vorgehen wird durch den Algorithmus im Anhang A, Abbildung 10 beschrieben).

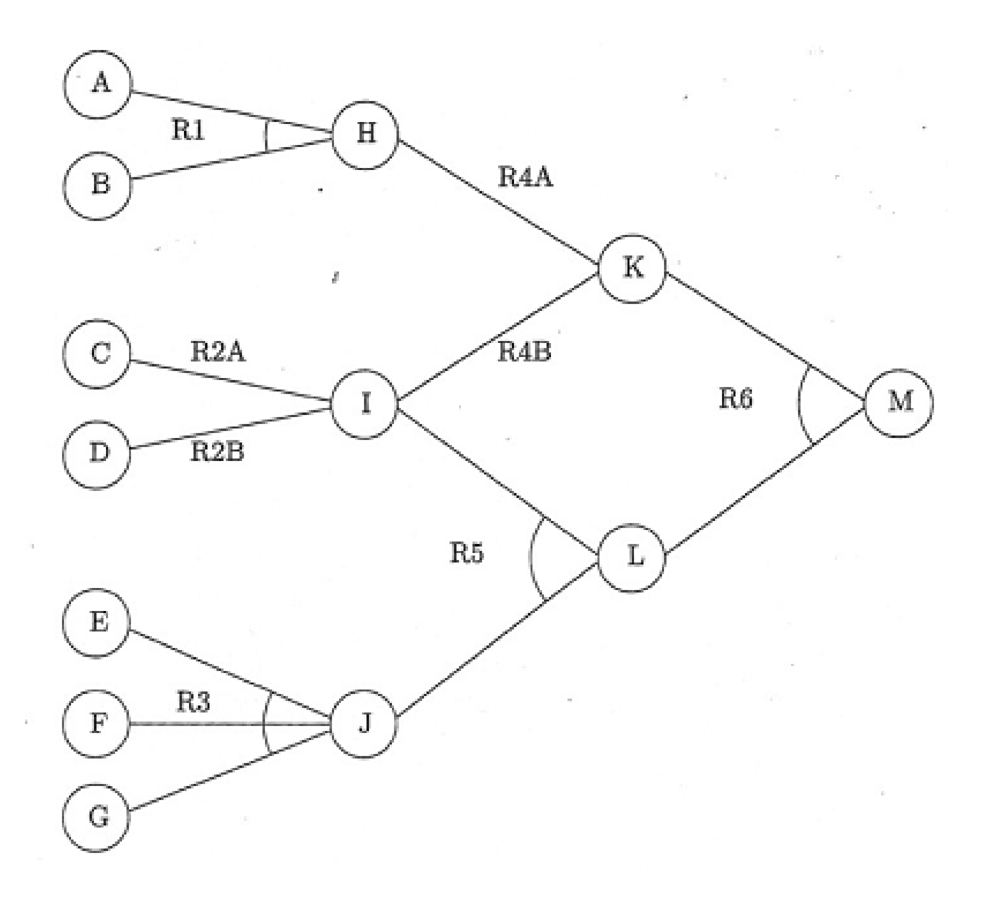


Abbildung 3. Regelnetzwerk eines regelbasierten Systems; die Kreisbögen symbolisieren konjunktive Verknüpfungen [5]

3.4 Rete Algorithmus

Bei naiver Implementierung der Regelinstanziierung müssen in jedem Schleifendurchlauf alle Eingaben mit den Bedingungsteilen aller Regeln verglichen werden. Das wird aber sehr problematisch bei großen Regelnmengen sein. Eine Effizienzsteigerung kann erreicht werden durch:

- **Einschränkung der zu überprüfenden Datenelemente:** Bei jeder Eingabe bzw. Regelanwendung wird jeweils nur ein kleiner Teil des Arbeitsspeichers verändert; nur dieser muss betrachtet werden.

- **Einschränkung der zu überprüfenden Bedingungen:** Die Bedingungsteile der Regeln sind im allgemein nicht disjunkt; gleichartige Test können zusammengefasst und damit der Testaufwand reduziert werden.

Auf der Grundlage dieser Überlegungen liefert der Rete-Algorithmus[5] eine effiziente Implementierung des Matching bei Vorwärtsverkettung mit Teilbedingungen und der Instanziierung von Regeln. Der Rete-Algorithmus wird in vielen regelbasierten Systemen verwendet, in denen es auf Effizienz ankommt, wie z.B. Jess, lLog, JRules, IBM Common-Rules[20].

Rete baut einen Baum aus Knoten verschiedenen Typen auf [10][6]:

Prämissenknoten: Diese Knoten stellen eine Prämisse dar.

Verbindungsknoten: Enthält eine Regel mehr als eine Prämisse, so werden die Verknüpfungen zwischen jeweils zwei Prämissen durch einen Verbindungsknoten dargestellt. Das Ergebnis eines solchen Knotens ist genau dann wahr, wenn beide Prämissen erfüllt sind.

Aktionsknoten: Dieses Ergebnis am Ende dient dann als Eingabewert für den Aktionsknoten, der die Aktion der Regel repräsentiert.

Beim ersten Evaluierungsdurchgang werden alle Prämissenknoten berechnet, und die Ergebnisse werden gespeichert. Ab dem zweiten Durchgang werden dann nur die Prämissen berechnet, deren Fakten sich geändert haben. Ändert sich damit auch das Ergebnis der Prämisse, wird das alte Ergebnis damit überschrieben. Ein Evaluierungsdurchlauf gehrt in dieser Weise:

– Ist eine Prämisse erfüllt!?. So gibt der Prämissenknoten sein Ergebnis an den folgenden Knoten weiter. Nun gibt es zwei Möglichkeiten:

1. Ist der folgende Knoten ein Aktionsknoten, so besitzt dieser Aktionsknoten nun eine gültige Prämisse als Eingang und somit wird die Aktion ausgeführt-
2. Ist der folgende Knoten ein Verbindungsknoten (Siehe Abbildung 4), so liegt nun auf einem der beiden Eingänge ein gültiges Ergebnis. Wenn auf dem zweiten Eingang nun ebenfalls schon ein gültiges Ergebnis liegt, sind beide Eingangsprämissen erfüllt, und damit ist auch das Gesamtergebnis dieses Knotens erfüllt.

3.5 Rule Engines

Im vorherigen Anschnitt wurden die Grundlagen von Regeln und regelbasierten Systemen erläutert. Es wurde ebenfalls erläutert, dass für die Verarbeitung von Regeln ein Interpreter benötigt wird, der die Regeln einlesen und durch die Mechanismen der Inferenz und der Konflikt-Lösung benötigte Informationen ableiten kann. Der Regelinterpreter in regelbasierten Systemen ist auf die Lösung bestimmtes Problem spezialisiert. Da in der Praxis die Unternehmen ihre Rahmenbedingung immer schnell ändern, war es in letzten Jahren nötigt, neue

[5] Der Rete - Algorithmus ist ein netzwerkbasierter Algorithmus und wurde von Charles Forgy entwickelt [20].

[6] Der Baum wird hier umgedreht, weil der Baum mit den Prämissenknoten als Blätter beginnt und im Aktionsknoten als Wurzel endet.

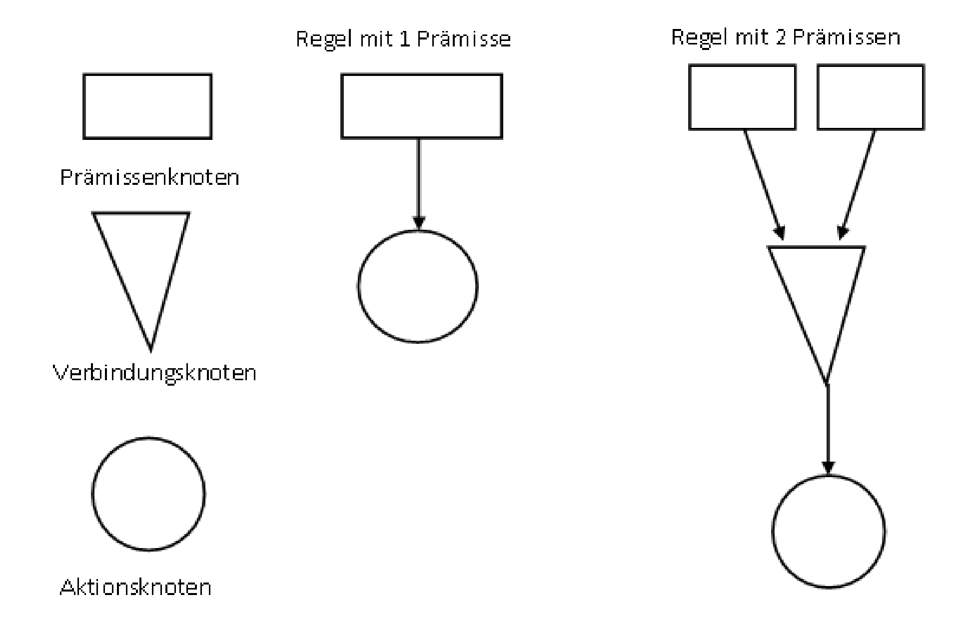


Abbildung 4. Knotentypen beim Rete-Algorithmus [20]

Regelinterpreter zu entwickeln, die die Prinzipien von Organic Computing verwenden und universell verwendbar sind. Solche Interpreter wurden *Rule Engines* genannt [2].

Rule Engines sollte freie Definitionen von Regeln und deren Anwendungen auf beliebiges Wissen ermöglichen. Der erste Versuch in dieser Richtung war das Expertensystem MYCIN[7] [15]. Der wichtigste Schritt in dieser Richtung war das System CLIPS[8], das eine universelle Regelsprachen bot, mit denen Expertensysteme für unterschiedliche Anwendungsbereiche realisiert werden konnten [8]. CLIPS kann als Vorläufer heutiger Rule Engines angesehen werden und aus CLIPS ist eine der heute erfolgreichen Rule Engines „Jess" (siehe Abschnitt 3.6) hervorgegangen. In diesem Abschnitt werden die Bestandteile Einer Rule Engine dargestellt und es wird erläutert, wie die Aufgaben in einer Rule Engine verarbeitet werden, wie die Regeln definiert werden, wie sie auf eine Datenbasis angewendet werden können.

[7] MYCIN wurde 1972 von der Standford University entwickelt und zur Diagnose von Infektionskrankheiten und deren Therapie durch Antibiotika eingesetzt.

[8] Abkürzung für C Language Integrated Production System. CLIPS wurde 1984 am Johnson Space Center der NASA entwickelt und wurde 1986 öffentlich zugänglich [8]

3.5.1 Bestandteile eines Rule Engines

Auf dem Markt gibt es verschiedene Rule Engine Implementierungen. Der Unterschiede liegen z.B. in den Strategien oder in den Algorithmen zur Konfliktlösung. Dennoch ähneln sich fast alle Produkte in ihrer Prinzipiellen Funktionsweise und verwenden fast die gleichen oder ähnlichen Begriffe zur Bezeichnung der jeweiligen Bestandteile.

Regeln und Regelbasis Wie im Abschnitt 3.1 erläutert wurde, stellen Regeln eine „Wenn dann“ Aussage dar. Sobald das erste Teil (Prämisse oder die Bedingung) einer Regel erfühlt, wird das zweite Teil (die Konsequenz oder die Aktion) der Regel ausgeführt. Als Beispiel für Regel aus der Praxis:

(Wenn der Gesamtpreis der Bestellung größer als 200 Euro ist, dann gewähre dem Kunden 15% Rabatt)

Alle Regeln des Systems werden in die Regelbasis (rule base) enthalten und die Definition der Regeln erfolgt mit Hilfe einer Regelsprache.

Fakten stellen die Datengrundlage für die Rule Engine dar. Fakten werden zur Erstellung der Prämisse einer Regel verwendet. D.h. die Prämissen sind jeweils Abfragen auf einen einzelnen Fakt. In obigem Beispiel stellt der Gesamtpreis den Fakt dar, auf dessen Grundlage die Prämisse *(Ist der Gesamtpreis größer als 200 Euro?)* erstellt wird

Aktionen stellen die Konsequenzen der Regeln dar. Die Konsequenz des obigen Beispiels lautet *(Gewähre dem Kunden 15% Rabatt)*. Diese Konsequenz berechnet im einfachsten Fall nur den Gesamtpreis der Bestellung neu und erzeugt damit einen neuen Fakt. Aber meistens führen die Aktionen zu komplexeren Operationen.

Arbeitsspeicher (Working Memory) Wie im Anschnitt 3.2 gezeigt wurde, enthält der Arbeitsspeicher Menge von Fakten, auf die das Regelset angewendet werden soll. Der Arbeitsspeicher ist ein wichtiger Bestandteil einer Rule Engine, weil auf ihn viele wichtige Operationen angewendet werden, wie z.B. hinzufügen, entfernen und modifizieren von Fakten und auch Regel feuern.

Regelinterpreter entscheidet, welche Regeln auf grund der gegeben Fakten und Bedingungen angewendet werden (gefeuert werden) und der Interpreter kann die Entsprechenden Aktionen starten. Aus diesem Grund, ist der Regelinterpreter der Kern einer Rule Engine. Die Komplette Wissensbasis, d.h. die Fakten und die Regeln, muss dem Regelinterpreter zur Verfügung gestellt werden.

Zusammenspiel dieser Komponenten Wie es in diesem Kapital erläutert wurde, dass es verschiedene Implementierungen von Rule Engines gibt, aber der Ablauf bei Einsatz einer Rule Engine ist grundsätzliche derselbe und wie folgendes [2]:

– Regel Laden

– Neuen Arbeitsspeicher erstellen

– Fakten in den Arbeitsspeicher hin fügen
– Rule Engine Starten

Das Zusammenspiel der Komponenten einer Rule Engine und diesen Ablauf ist in Abbildung 5 dargestellt.

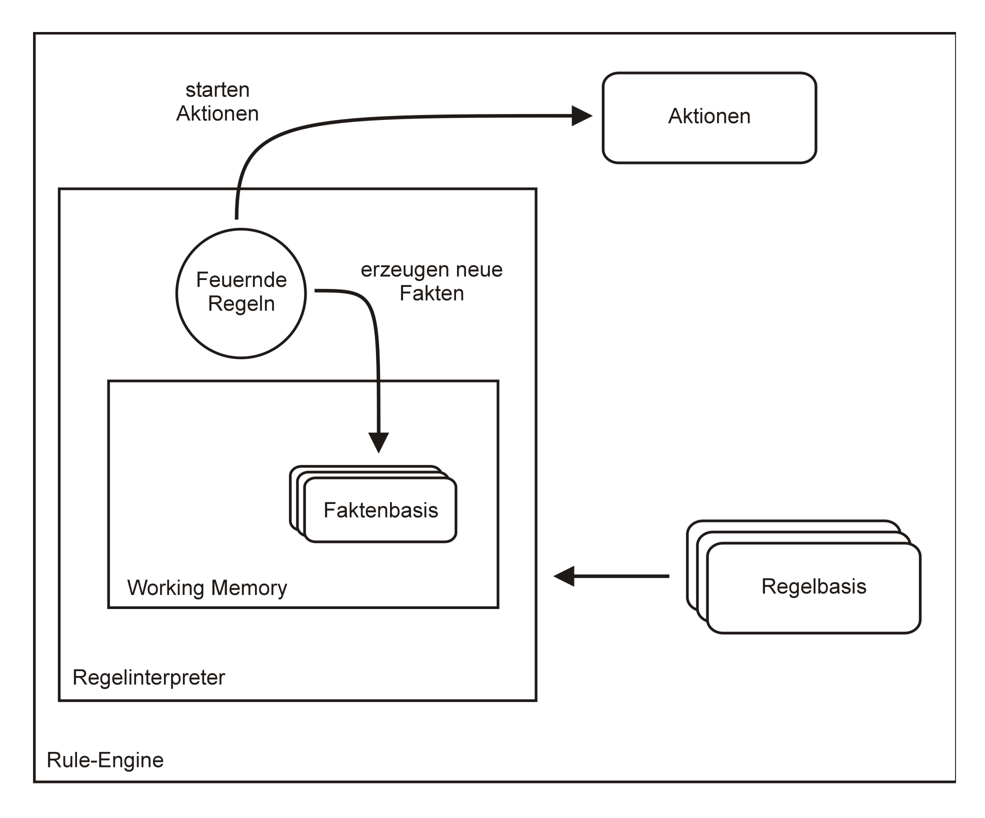


Abbildung 5. Bestandteile einer Rule Engine [2]

3.6 Rule Engines in der Praxis

In jedem Unternehmen existieren Regelungen für die ablaufenden Prozesse, wie z.B. für die Vermarktung, für das Personal usw. Diese Geschäftsregelungen sind schon vorhanden, sie sind jedoch oft nicht einheitlich formuliert. Der Einsatz von Rule Engine bzw. Business Rule Engine[9] erhöht die Flexibilität der IT-Systeme. Die Rule Einige wird die vorhandenen Regelungen explizit machen

[9] Business-Rule-Engine (BRE) ist eine technische Softwarekomponente als Bestandteil eines Business-Rule-Management-Systems (BRMS), die eine effiziente Ausführung von Geschäftsregeln bzw. Business-Rules ermöglicht [21].

und eine automatisierte Verarbeitung von diesen Regelungen führen und die in einer Form von Wenn-Dann-Regeln festlegen. Da die Rolle von Rule Engines so bestechend ist, sind eine Vielzahl von Rule Engines entwickelt worden teils kommerziell, teils im Opensource Bereich. Die Rule Engines können nach den unterstützten Inferenzstrategie unterschieden werden und wie folgend:

3.6.1 Vorwärtsverkettende Systeme Die meisten Rule Engines in der Praxis unterstützen diese Art von Verkettung. Als Beispiel stellt JRuls[10] die teuerste und leistungsfähigste Rule Engine am Markt. JRule bietet Schnittstellen für Java und .NET und ist sowohl für Microsoft Windows als auch für Unix/Linux Umgebung verfügbar. Die Regel in JRule können in der natürlichen Sprache Englisch dargestellt werden und auch in IRL (Ilog Rule Language).

Eine Alternative für diese Art von Verkettung stellt das Open Source JBoss Rule/Drools (sehe Kapitel 4) dar. JBoss bietet eine Java-Basierte Business Rule Engine, die effizient und schnell ist. JBoss ist nur eine Komponente zur Regelverarbeitung und stellt keine vollständiges Business Rules Management System dar. Die Programmieroberfläche ist in der Java-Editor Eclipse, in dem das Drolls-Plugin integriert ist. Die Abfrage einer Variablen eines Objekts wird durch die Drolls Sprache ausgedrückt (vgl. Kapitel 4).

3.6.2 Rückwärtsverkettende Systeme Am Markt gibt´s nicht so viele Rule Engines, die die Rückwärtsverkettung unterstützen aber Mandarax[11] stellt die bekannteste Rule Engine dar, die diese Art von Verkettung unterstützt. Mandarax ist eine Open Source Java-Klassenbibliothek für regelbasierte Systeme. Mandarax unterstützt als erste Rule Engine am Markt RuleML. Im Kapital 4 werden RuleML und Drools näher betrachtet.

3.6.3 Hybride Systeme Jess[12] ist die bekannteste Rule Engine am Markt für Java Platform. Jess arbeitet grundsätzlich als Vorwärtsverkettendes System beherrscht aber, wenn benötigt, auch Rückwärtsverkettung. Jess hat bei der Regelbeschreibung seine eigene Wege: Zur Deklaration der Regeln wird die an die Syntax von Lisp[13] angelehnte Beschreibungssprache „CLIPS" verwendet, die im Vergleich zu herkömmlichen Programmiersprachen bzw. Markup-Sprachen wie XML sehr gewöhnungsbedürftig ist, was die Einführung im Unternehmen und die Akzeptanz deutlich erschweren könnte [6].

[10] Mehr Informationen unter : http://www.ilog.com/products/jrules/.

[11] Mehr Informationen unter http://mandarax.sourceforge.net/ .

[12] Mehr Informationen unter : http://herzberg.ca.sandia.gov/ .

[13] LISP steht für List Processing und ist eine Familie von Programmiersprachen, die 1958 erstmals spezifiziert wurde und am Massachusetts Institute of Technology (MIT) in Anlehnung an den Lambda-Kalkül entstand [22].

4 Drools und RuleML im Vergleic

In Diesem Kapitel wird beschrieben wie die konkrete Implementierung von Regel im RuleML und Drools erfolgen kann. Anhand von konkretem Beispiel, werden wir sehen wie die beiden Regelsprache sich gegenüber stellen.

4.1 RuleML

RuleML ist die Abkürzung für Rule Markup Language (Beschreibungssprache für Regel). RuleML stellt einen Ansatz zur standardisierten Beschreibung von Regeln auf Basis von XML dar und bietet Elemente zur Implementierung von verschiedenen Logik- und Regelsystemen. Somit ist RuleML als Transfermedium zwischen Rule-Engines und anderen Systemen geeignet.

4.1.1 RuleML Design Das aktuelle RuleML-Design ist eine Hierarchie von Regeln wie in der Abbildung 6. Die Wurzel dieser Hierarchie bildet Rules. Dann verzweigt sie sich in Reactionen Rules und Transfomation Rules, die selbst sich weiter unterteilen [1].

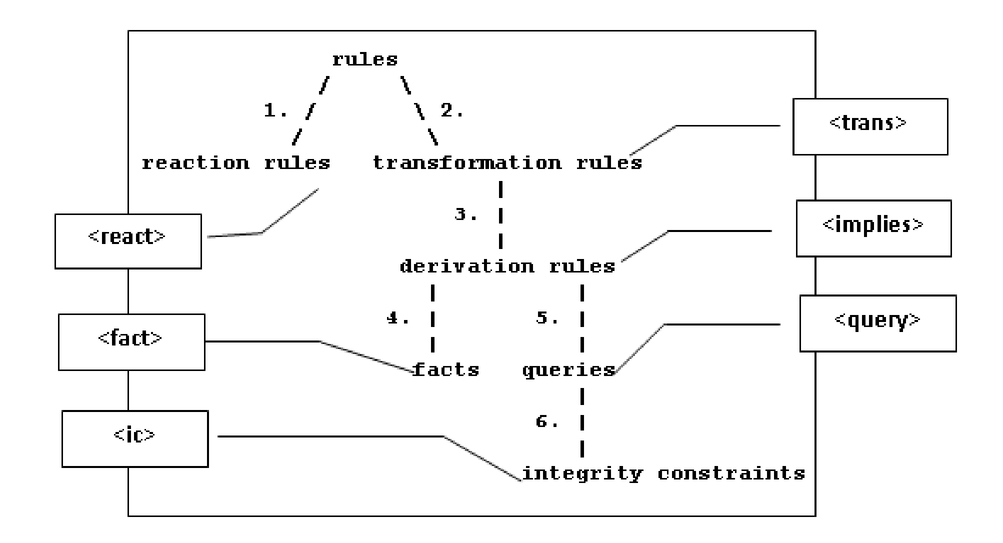


Abbildung 6. Regelhierarchie in RulesML [1]

Reaktionsregeln Bestehen aus einem Ereignis, einer Bedingung, einer Aktion, und möglicherweise einer Nachbedingung. Z.B.: *Wenn X ein Mietauto ist, nicht reparaturbedürftig ist, und nicht anderem Mieter zugeweisen ist, dann gilt dass es zur Verfügung steht*

Transformationsregeln Bestehen aus einer Bedingung, einer Konklusion und einer Transformationsausgabe. Z.B.: *Transform die lange Beschreibung eines Buches in eine kurze Beschreibung, die nur den Titel dieses Buches enthält*

Ableitungsregel*n* Ableitungsregeln sind sie Schlussregeln und bestehen aus einer oder mehreren Bedingungen und einer Konklusion. Z.B.: *Ein Auto ist zur Verfügung steht, wenn es ein Mietauto ist, nicht reparaturbedürftig ist, und nicht anderem Mieter zugewiesen ist*

Integrationsregeln Bestehen nur aus Bedingungen. Z.B: *Ein Mietautofahrer muss mindestens 22 Jahre alt sein*

4.1.2 Syntax Beispiel Bei der Beschreibung von Fakten und Regeln ist es möglich, sich weitergehend an der logischen Programmsprache Prolog zu orientieren. In RuleML werden die Regeln in einer Baumstruktur geschrieben wie im folgenden Beispiel für die Ableitungsregeln (Derivation Rules): *The discount for a customer buying a product is 5 percent if the customer is premium and the product is regular.*

In dieser Regel gibt es zwei Prämissen du eine Konklusion. Die zwei Prämissen werden durch die Einführung eines expliziten <And > gelöst. Im RuleML 0.91 steht body für die Prämissen, head für die Konklusionen und <Atom > für die logischen Ausdrücke. Die RuleML Notation für die Regel hat die folgende Gestalt [6]:

```
1  <Implies>
2    <head>
         <Atom>
4          <Rel>discount</Rel>
        <Var>customer</Var>
6       <Var>product</Var>
        <Ind>5.0</Ind>
8        </Atom>
       </head>
10     <body>
         <And>
12         <Atom>
          <Rel>premium</Rel>
14        <Var>customer</Var>
        </Atom>
16      <Atom>
          <Rel>regular</Rel>
18        <Var>product</Var>
        </Atom>
20       </And>
       </body>
22   </Implies>
```

Syntax für weitere Regelarten in RuleML kann wie folgendes dargestellt werden:

– **Transformationsregel**

```
<trans>
<_head>Konklusion</_head>
<_body><and>Bed1...BedN</and></_body>
<_foot>Transformationsausgabe</_foot>
</trans>
```

– **Fakten**

```
<fact><_head>Konklusion</_head></fact>
Oder
<implies>
<_head>Konklusion</_head>
<_body><And></And></_body>
</implies>
```

– **Anfragen**

```
<query>
<_body><And>Bed1...BedN</And></_body>
</query>
```

– **Integritätsregeln**

```
<ic>
<_body><And>Bed1...BedN</And></_body>
</ic>
```

– **Reaktionsregeln**

```
<react>
<_event>Auslöser</_event>
<_body><And>Bed1...BedN</And></_body>
<_head>Aktion</_head>
</react>
```

4.1.3 Einsatz von RuleML RuleML wird von einigen Rule-Engines, wie zum Beispiel Mandarax, unterstützt. Besonders offene Anwendungen legen Wert auf transparente Datenformate. Der Einsatz von RuleML ist sinnvoll für die Modellierung von Regelsystemen, da sie einen effektiven und formalen Rahmen für die Implementierung bieten. Ferner ist die Verwendung von RuleML eine mögliche Basis für die Entwicklung von Import- und Export-Formaten zwischen verschiedenen Systemen und somit ein Ansatz für eine standardisierte Speicherung von Geschäftsregeln.

4.2 Drools

Drools ist ein Open-Source vorwärtsverkettendes Rule Engine. Drools stellt ihre Notation auf der Basis der Metasprache XML mit eingebettetem Java-Code in eine gute Kombination [23]. Als relativ junges Projekt wird es zum größten Teil von Entwicklern zum Einsatz in eigenen Java-Projekten bzw. zur Evaluierung der Möglichkeiten genutzt. Diese Entwickler sind sehr vertraut mit Java, das darüber hinaus auch den kompletten Sprachumfang zur Formulierung von Bedingungen und Aktionen bereit stellt. Für XML gelten dabei ähnliche Argumente. Die Vorteile auf Seiten der Entwickler liegen ebenfalls auf der Hand: Sowohl die Interpretation von Java-Code als auch der Umgang mit XML-Dateien kann über fertige Komponenten abgewickelt werden. Damit entfällt die Entwicklung eigener Komponenten zur Erfassung und Verarbeitung der Regeln.

4.2.1 Drools Design Die Grundstruktur eines DRL-Dokuments ist folgende:

- das Wurzelelement heißt rule-set und muss ein Attribut name besitzen kann aber auch ein Attribut description haben.
- das Wurzelelement kann beliebig viele rule-Subelemente besitzen, die wiederum ebenfalls ein Attribut name besitzen müssen; der Wert des name-Attributs muss im ganzen Dokument eindeutig sein.
- ein rule-Element kann beliebig viele Parameter-Subelemente besitzen, wobei ein Parameter-Element ein Attribut identifier haben muss. Der Inhalt eines Parameter- Elements ist in der Regel ein class-Subelement, dessen Inhalt der qualifizierte Klassenname des Parametertyps angibt.

ein rule-Element kann beliebig viele condition-Subelemente (Bedingungen)haben und muss genau ein consequence-Subelement (Konklusionen)besitzen.

4.2.2 Regelattributen no-loop
typ: Boolean.
default Wert: false.
Es ist möglich, dass die Konsequenz einer Regel dazu führen würde, dass diese Regel nochmals evaluiert wird. Mit diesem Parameter kann man verhindern, dass die Regel mehrmals evaluiert wird.

salience
typ: integer.
default Wert: 0.
Salience ist eine Art Priorität Wert. Regeln mit einem höheren Saline Wert werden eher ausgeführt als solche mit niedrigem Wert. Man kann hier einen positiven sowie auch negativen Wert angeben.

activation-group
typ: String.
default Wert: NULL.

Mit dem Festlegen einer Activation Group können mehrere Regeln zu einer solchen Gruppe zusammengefasst werden. Die erste Regel dieser Gruppe die ausgeführt wird schickt allen anderen der Gruppe ein Signal, sodass diese nicht mehr ausgeführt werden. **Beispiel:**
rule ´´meine Regel´´
salience 42
activation-group wichtig
when
...

4.2.3 Syntax Beispiel In Drools steht rule für Regel, condition für Bedingung/ Prämisse und consequence für Konklussion/ Aktion. Für dasselbe Beispiel von RuleML sollte die Regelbeschreibung von dieser Regel mittels XML so ausschauen.

```
<rule name="the discount for a customer?">
    <parameter identifier="customer">
            <class>com.camunda.Customer</class>
      </parameter>
      <condition>
                customer.status ().equels (''premium'')
        </condition>
        <condition>
            customer.product().equels (''regular'')
        </condition>
        <consequence>
            apply5PercentDiscountAction(customer, drools) ;
      </consequence>
</rule>
```

Drools Rule Language (DRL) unterstützet natürlich Java und außerdem einige Skriptsprachen wie z. B. Python und Groovy. Für jede dieser Sprachen gibt es in DRL einen eigenen Namensraum. Setzt man Java ein, sieht ein condition-Element so aus:
<java:condition>...</java:condition>
Dabei ist der Inhalt des Elements natürlich ein Java-Ausdruck mit einem Booleschen Wert. Und unter Java sieht ein consequence-Element so aus:
<java:consequence>... </java:consequence>
wobei der Inhalt ein Block von Javacode ist.

4.2.4 Einsatz von Drools Drools unterstützt den Standard JSR-94 des Java Community Prozesses[14] definiert eine Schnittstelle zu einer Rule-Engine, die unabhängig von der konkreten Implementierung der Rule-Engine (also unabhängig vom eingesetzten Rule-Engine-Produkt) machen soll [13].Im 2006 hat sich das Drools-Projekt dem Unternehmen *JBoss Inc.* angeschlossen, unter anderem Entwickler des populären gleichnamigen Applikation Servers [[23]. Drools ist eigentlich eine Klassenbibliothek in Java, die sich einfach und problemlos in eigene

[14] Der Java Community Process (JCP) definiert in Kooperation mit Unternehmen, Hochschulen und Organisationen Standards für die Programmiersprache Java.

Java-Anwendungen integrieren lässt. Außer den Kernelementen der Inferenzmaschine sind aber keine weiteren Werkzeuge enthalten. Am Markt sind auch noch keine ausgereiften Werkzeuge erhältlich. Drools bietet die Möglichkeit, Regeln in einem proprietären XML-Format zu erfassen oder mittels XSLStylesheet eigene Fachsprachen zu entwickeln [13].

4.3 Vergleich

In vorigen Abschnitten haben wir einen Überblick über RuleML und Drools gewonnen. Die Beiden Regelsprachen sind Open Source. Regeln können mit RuleML in natürlicher Sprache, in irgendeiner formalen Darstellung oder in einer Kombination von beiden darstellt werden. Es gibt auch viele Tools, Editoren und Translator die RuleML unterstützen. Aber trotzdem lassen die Regeln sich mit Drools leichter darstellen. Drools wurde auch als Plugin in JBoss integriert.

Da die Verwendung von Rete Algorithmus die Performance erhört, sollte Drools bessere Performance als RuleML haben. Schließen von neuem Wissen aus vorhandenem Wissen und gegebenen Regeln geschieht bei RuleML nach Rückwärtsverkettungsstrategie, wobei bei Drools geschieht es nach Vorwärtsverkettungsstrategie .

RuleML stellt eine Initiative von verschiedenen Organisationen aus Industrie und lehre dar, die einen künftigen Standard einer Beschreibungssprache für Regeln definieren will. Unter dem Gesichtspunkt, dass Regeln im Systemtechnischen Sinne allgemein gültige Regeln darstellen sollen, kann ein solcher Standard zu einem wichtigen Werkzeug bei der Verwendung von regelbasierten Systemen werden. Dann könnte auch der Austausch von Rule-Sets zwischen verschiedenen Systemen verschiedener Hersteller greifbar (da kostengünstig) werden. Wobei stellt Drools eine eigene Rule Engine dar, die mittlerweile in einer stabilen und ausgereiften Version zur Verfügung steht. Dieses System ist eigentlich eine Klassenbibliothek in Java, die sich einfach und problemlos in eigene Java-Anwendungen integrieren lässt.

5 Fazit

– Selbstorganisation fokussiert sich auf die Systemumgebungsgrenzen, wenn die Grenze zwischen inneren und äußeren als Teilen betrachtet werden.

– Im Gegenteil dazu nimmt der Prozess des Emergenz Plätze an der Grenze zwischen dem System und seinen Bestandteilen

– Ein Regelbasiertes System ist ein Wissensbasiertes System in dem regelbasiertes Schließen stattfindet

– Regeln sind formalisierte Konditionalsätze der Form: if A then B.

– Es gibt zwei Strategien zum Schließen von neuem Wissen: Vorwärtsverkettung und Rückwärtsverkettung.

– Rete Algorithmus bietet eine effiziente Implementierung für Vorwärtsverkettung. Es erhöht auch die Performance des Systems.

– Ein Rule Engine ist ein raffinierter Interpreter für if/then-Aussagen.

– Die Verwendung einer Rule Engine bringt sicherlich in einigen Bereichen viele Vorteile. Es ist aber immer abzuwägen, was genau in Rules abgebildet werden soll.

– Gerade dort, wo Skalierung wichtig ist, viele if-else-Bedingungen auftreten und schnell Regeländerungen umgesetzt oder ausgeführt werden müssen, ist eine Rule Engine ideal.

– RuleML stellt einen Ansatz zur standardisierten Beschreibung von Regeln auf Basis von XML dar und bietet Elemente zur Implementierung von verschiedenen Logik- und Regelsystemen.

– Das aktuelle RuleML-Design ist eine Hierarchie von Regeln und Die Wurzel dieser Hierarchie bildet Rules. Dann verzweigt sie sich in Reactionen Rules und Transfomation Rules, die selbst sich weiter unterteilen.

– Drools ist ein Open-Source vorwärtsverkettendes Rule Engine. Drools stellt ihre Notation auf der Basis der Metasprache XML mit eingebettetem Java-Code in eine gute Kombination.

– Drools ist eigentlich eine Klassenbibliothek in Java, die sich einfach und problemlos in eigene Java-Anwendungen integrieren lässt.

Literatur

[1] J. Baydoun. Regelsprachen, institut für informatik, fh-berlin, 2005. *Unter:* `http://www.ag-nbi.de/lehre/05/S_MOD/Regelsprachen_180505.pdf`, Zugriff 25.02.08.

[2] H. Beck. Einsatz von Rule-Engines zur flexiblen Wissensverarbeitung im betrieblichen Umfeld. *Bachelorarbeit an Fachhochschule Darmstadt*, 2005.

[3] C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme*. Vieweg Verlag, p71, 2006.

[4] C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme*. Vieweg Verlag, p78, 2006.

[5] C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme*. Vieweg Verlag, p81, 2006.

[6] H. Boley. The ruleml family of web rule languages, university of new brunswick, canda, 2006. *Unter: http://2006.ruleml.org/slides/RuleML-Family-PPSWR06-talk-up.pdf*, Zugriff 25.02.08.

[7] A. Champandard. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors, Kapitel 11 „Rule-Based Systems“*. New Riders, 2003.

[8] CLIPS. CLIPS, A Toll for Building Expert System. *Unter:http://www.ghg.net/clips/CLIPS.html*, Zugriff 25.02.08.

[9] A. Depot. Methods of rule-based systems. *Unter: http://ai-depot.com/Tutorial/RuleBased-Methods.html*, Zugriff 25.02.08.

[10] E. Friedman-Hill. Jess. The Rete Algorithm, Sandia National Laboratories, 2000.

[11] J. Fromm. Ten Questions about Emergence. *Arxiv preprint nlin.AO/0509049*, 2005.

[12] S. H., U. T., and C. Müller-Schloer. Antrag auf Einrichtung eines neuen DFG-Schwerpunktprogramms. *Unter: http://www.organic-computing.de/spp*, Zugriff 25.02.08.

[13] JBoss. Jboss.org community driven. *Unter: http://labs.jboss.com/*, Zugriff 25.02.08.

[14] H. Kasinger. Ein MDA-basierter Ansatz zur Entwicklung von Organic Computing Systemen, 2005. *Unter: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/vs/publikationen/reports/2005-08/*, Zugriff 25.02.08.

[15] C. Merz. Entwicklung eines computerspiels mit dem schwerpunkt auf künstlicher intelligenz. *Diplomarbeit an Hochschule Ravensburg-Weingarten*, 2006.

[16] C. Müller-Schloer, C. von der Malsburg, and R. Würt. Organic Computing. *Informatik-Spektrum*, 27(4):332–336, 2004.

[17] Wikipedia. Wikipedia-Die freie Enzyklopädie. *Unter:http://de.wikipedia.org/wiki/Wissensbasiertes_System*, lZugriff 25.02.08.

[18] Wikipedia. Wikipedia-Die freie Enzyklopädie. *Unter:http://de.wikipedia.org/wiki/Regelbasiertes_System*, Zugriff 25.02.08.

[19] Wikipedia. Wikipedia-Die freie Enzyklopädie. *Unter: http://de.wikipedia.org/wiki/ Expertensystem*, Zugriff 25.02.08.

[20] Wikipedia. Wikipedia-Die freie Enzyklopädie. *Unter: http://en.wikipedia.org/wiki/Rete_algorithm*, Zugriff 25.02.08.

[21] Wikipedia. Wikipedia-Die freie Enzyklopädie. *Unter: http: //-de.wikipedia.org/wiki/ Business-Rule-Enginehttp*, Zugriff 25.02.08.

[22] Wikipedia. Wikipedia-Die freie Enzyklopädie. *Unter: http://de.wikipedia.org/wiki/LISP*, Zugriff 25.02.08.

[23] Wikipedia. Wikipedia-Die freie Enzyklopädie. *Unter: http://en.wikipedia.org/wiki/Drools*, Zugriff 25.02.08.

Anhang A

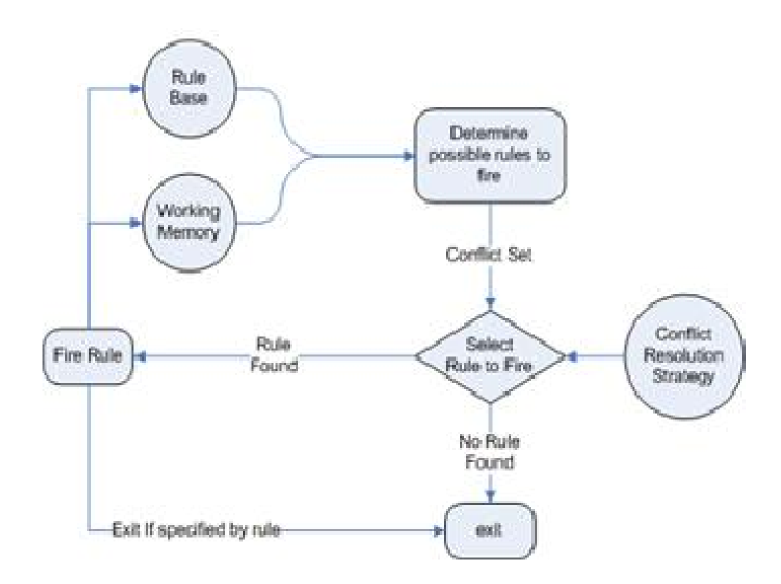
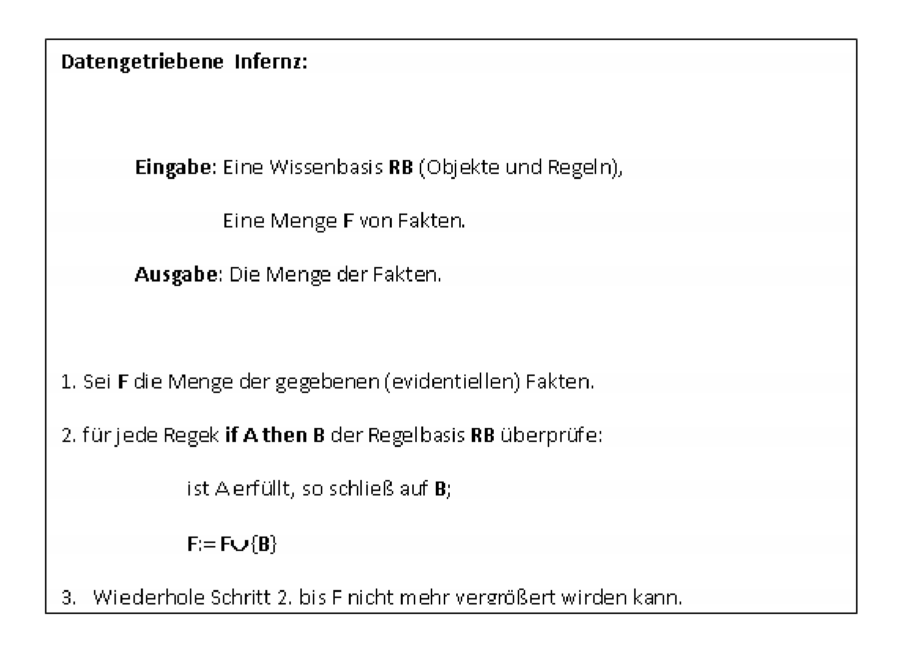


Abbildung 7. Vorwärtsverkettung [9]



Datengetriebene Infernz:

Eingabe: Eine Wissenbasis **RB** (Objekte und Regeln),

Eine Menge **F** von Fakten.

Ausgabe: Die Menge der Fakten.

1. Sei **F** die Menge der gegebenen (evidentiellen) Fakten.
2. für jede Regek **if A then B** der Regelbasis **RB** überprüfe:

   ist A erfüllt, so schließ auf **B**;

   F:= **F**$\cup${**B**}

3. Wiederhole Schritt 2. bis F nicht mehr vergrößert wirden kann.

Abbildung 8. Algorithmus zur datengetriebenen Inferenz [4]

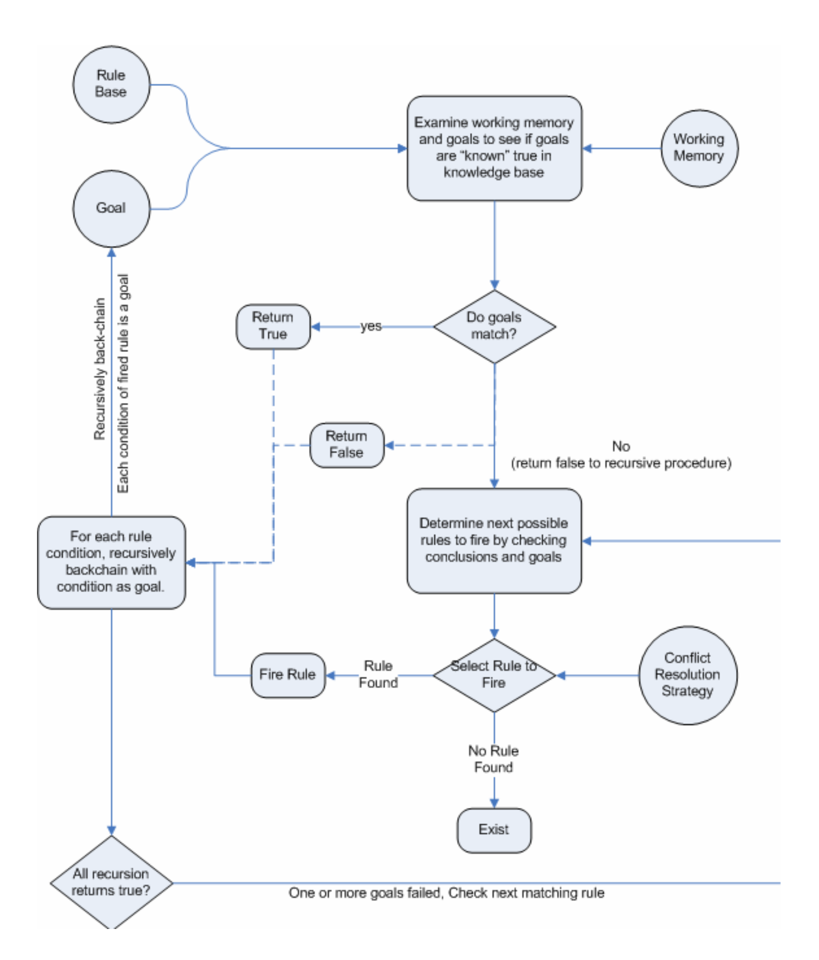


Abbildung 9. Rückwärtsverkettung [9]

```
Zielorientierte Infernz:

    Eingabe: Eine Wissensbasis RB (Objekte und Regeln),
             Eine Menge F von Fakten, eine Liste von Zielen
             (atomaren Anfragen) [q1,.... qn]
    Ausgabe: yes, falls alle qi ableitbar sind, sonst no.

BACKCHAIN ([q1,.... qn])

    BACKCHAIN ([q1,.... qn])

            if n := 0 then returen (yes) ;
            if q1∈ F
              then BACKCHAIN ([q2,.... qn])
              else for each Regel p1∧....∧pm →aus RB mit q1=
q2
               do if BACKCHAIN ([p1..., pm,q2,...,qn]) = yes
                   then return (yes)
               endfor
            endif
            return (no)
```

Abbildung 10. Algorithmus zur zielorientierten Inferenz [4]

Policies im Organic Computing

Andreas Meixner

Universität Augsburg
andreasmeixner@gmx.de

Zusammenfassung Der Konfigurations- und Managementaufwand moderner IT Systeme nimmt ständig zu. Um in Zukunft noch leistungsfähigere und komplexere Systeme zu ermöglichen, die mit heutigen Mitteln, auf Grund von zu hohem Managementaufwand, nicht betrieben werden könnten bedarf es neuer Techniken. Zwei dieser Techniken sind Organic Computing und Policy Based Management. OC Systeme sind Computersysteme die sich vor allem dadurch auszeichnen, dass sie ihr Verhalten in vielen Bereichen seblstständig entscheiden können. Durch diese Selbstständigkeit wird der Bedarf an menschlicher Interaktion wie Konfiguration und Anpassung an neue Voraussetzungen minimiert. Gleichzeitig besteht aber die Notwendigkeit das Verhalten der Systeme zu beeinflussen und zu lenken, um negatives Verhalten auszuschließen. Man denke hierbei an die zahlreichen Horrorvisionen von Science-Fiction Autoren wie beispielsweise die Kinoklassiker Terminator und Matrix. Eine Technik diese Diskrepanz zwischen Entscheidungsfreiheit und Kontrolle des Verhaltens zu überwinden ist die des Policy Based Managements. In dieser Arbeit soll beleuchtet werden, was Policies sind, wie sie für das Systemmanagement, insbesondere im bereich OC dienen können, und was heute bereits umgesetzt und anwendbar ist.

1 Was ist eine Policy?

Das Wort Policy ist aus dem Bereich der Politik, und bedeutet politischer Kurs, Richtlinie oder auch Regelwerk. Eine wichtige Eigenschaft der Policy ist, dass sie nicht immer streng eingehalten wird, sondern als eine Art Entscheidungshelfer dient. Beispielsweise sollte die Policy einer Partei folgende Regel beinhalten: „Wir werden alles dafür tun, dass es den Menschen in unserem Land gut geht.“ Vor einer Wahl kann es jedoch geschehen, dass die Partei diesen Vorsatz, zusammen mit eigentlich wichtigen - aber auch unpopulären - Maßnahmen hinten anstellt, um keine Wählerstimmen zu verlieren. In diesem Fall wurde also gegen die Policy verstoßen. Bei der nächsten Entscheidung die die Partei trifft wird die Policy jedoch wieder als Entscheidungshelfer herangezogen. Policies gibt es aber nicht nur in der Politik, sondern auch in allen anderen Gesellschaftsbereichen, wie z.B. die 10 Gebote des Christentums, die GoB in der Wirtschaft, oder dass sich Jogger grüßen wenn sich sich beim Laufen begegnen. Auch hier gilt, die Policy ist kein Gesetz, sondern eine Hilfe beim Treffen einer Entscheidung, und man kann sich jederzeit anders entscheiden als es die Policy vorgibt.

Sieht man sich in der Literatur zu diesem Thema um wird man schnell feststellen, dass der Begriff Policy sehr flexibel benutzt wird. So wird sowohl eine einzige Regel wie die oben genannten als Policy bezeichnet, aber auch ein ganzer Satz von solchen.

1.1 Umsetzung von Policies in der IT Heute

Die Umsetzung von Policies in der IT war bisher sehr starr. Es wurden Systeme entwickelt, die keine oder nur sehr wenige selbstständige Entscheidungen treffen konnten. Dadurch war sichergestellt, dass ihr Verhalten immer den Richtlinien entspricht. Ein entscheidender Nachteil ist allerdings, dass bei Änderungen an der Policy unter umständen das gesamte System verändert werden muss, um ihr angepasst zu werden. Außerdem ist die Entwicklung mit diesem Ansatz sehr aufwendig, da alle Zustände die ein System einnehmen kann bereits zur Entwicklungszeit bekannt seien müssen, um die Reaktion des Systems festzulegen. Bei der Größe heutiger IT-Systeme ist dies allerdings nicht mehr möglich. Daraus resultieren Fehler im Verhalten und die Notwendigkeit Anpassungen vorzunehmen.

Eine weitere Herausforderung moderner Softwaresysteme ist, dass mit zunehmender Größe neue Architekturen wie die 'komponentenbasierten Software' oder 'Service Oriented Architecture' Einzug gehalten haben. Diesen neuen Paradigmen is gemein, dass ein System nicht als ein einziges monolithisches Gebilde aufgebaut ist, sondern aus einer Vielzahl kleiner Bausteine besteht, die jeweils einen kleinen Teil der Gesamtaufgabe erfüllen. Der Aufwand für Konfiguration und Wartung immer hier drastisch zu, da nicht nur jede Komponente für sich konfiguriert werden muss, sonder auch das Zusammenspiel der Komponenten untereinander. Zudem sind solche System meist räumlich und logisch verteilt, so dass die Konfiguration noch weiter erschwert wird. Eine Gefahr die sich bei verteilten Systemen ergeben kann ist, dass unerwünschtes emergentes Verhalten auftritt. Darunter versteht man ein Verhalten, dass nicht aus der Betrachtung der einzelnen Teile ableitbar ist. Bei der üblichen Arbeitsteilung zwischen System-/Softwarearchitekten, die einen Überblick über das Gesamtsystem haben, aber die Komponenten nicht im Detail kennen, und Komponentenentwicklern, die ihre Komponente genau kennen, jedoch nicht das Gesamtsystem oder andere Komponenten, ist es unmöglich emergente Verhaltensweisen im Voraus zu erkennen.

1.2 Ein neues Konfigurationsmodell für die Zukunft

Eine Lösung für vile der oben genannten Probleme soll das Policy Based Management (kurz PBM) liefern. Die Grundidee ist, dass die Policies nicht mehr hart im Programmcode der Software codiert werden, sondern dem System in einer Konfigurationsdatei, Datenbank oder Ähnlichem direkt vom Systembetreiber mitgeteilt werden. Damit reduziert sich der Aufwand für die Softwareentwicklung, da ein großer Teil der Logik nicht mehr von den Entwicklern, sondern den Systembetreibern implementiert wird. Ein weiterer Vorteil ist, dass ein System bei dem die Funktionalität und Verhaltenssteuerung voneinander getrennt sind

sich während des laufenden Betriebes an neue anforderungen anpassen lässt. Es ist aber auch klar, dass PBM die heutigen Konfigurationsmechanismen nicht vollständig ersetzen, sondern nur ergänzen und flexiebler machen soll. Um den Pfad einer Log-Datei oder die URL einer Datenbank festzulegen sind die heute üblichen Name-Wert-Paare am geeignetsten. Allerdings sollen Policies diese Einstellungen , möglichst automatisiert, ändern und an die aktuelle Systemumgebung anpassen.

2 Policies in OC Systemen

2.1 Was versteht man unter OC

Das Organic Computing oder auch Bio Inspired Computing, versucht Prinzipien aus der Natur auf Computer zu übertragen. Dabei geht es nicht darum die natur so detailgetreu wie möglich zu kopieren, sondern Mechanismen zu verstehen und auf die Anwendung in der IT zu übertragen und anzupassen. Das wohl bekannteste Beispiel uns dem OC sind die Ameisenalgorithmen, die es erlauben für das NP-harte „Traveling Salesman“ Problem in extrem kurzer Zeit eine nahezu optimale Lösung zu errechnen. OC umfasst unter anderem die Bereiche Computerimmunologie, Schwarmverhalten, Genetische Algorithmen, und auch das von IBM propagierte Autonomic Computing(AC) ist eine Teilmenge des OC. AC hat sich zum Ziel gesetzt, dass sich IBM-Server in Zukunft nahezu vollkommen selbstständig verwalten. Zu diesem Zweck müssen sie die sogenannten Self-X-Eigenschaften umsetzten. Zu ihnen zählen Self-Configuration, Self-Healing, Self-Optimization und Self-Protection. Zusätzlich müssen diese Server Umgebungsbewusstsein (Contextawareness) aufweisen, da sie sich eben genau auf ihre Umgebung selbstständig einstellen sollen.

2.2 Aufbau eines OC Systems

Die grundlegende Architektur eines OC Systems besteht aus drei Teilen.

System : Das zu managende System.
Observer : Beobachtet das System und seine Umgebung.
Controller : Erhält vom Observer Informationen über den Systemzustand und legt die Reaktion des Systems fest, bzw. konfiguriert/optimiert es entsprechend.

Observer und Controller können wie in Abbildung 1 für das Gesamtsystem verantwortlich sein, oder es kann einen Observer bzw. Contriller für jeden Systemteil geben. Eine andere Möglichkeit sind hierarchisch angeordnete Observer/Controller, beispielsweise ein Observer für jeden Systemteil, und darübr ein Observer der diese Observer überwacht.

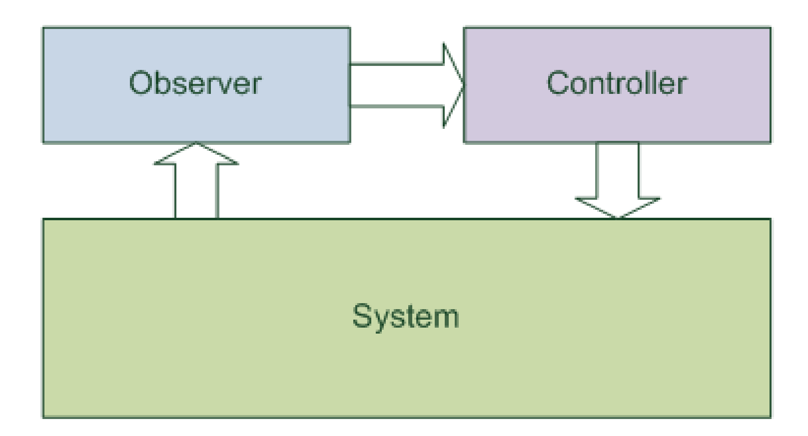


Abbildung 1. Observer-Controller-Architektur eines OC Systems

2.3 Verhaltenssteuerung in der OC Architektur

Da Policies genau dazu dienen Verhalten zu steuern, sind sie ein ideales Mittel für die Realisierung des Controllers ind der OC Architektur. Das Verhalten in diesen Systemen kann man, wie bei allen verteilten Systemen, in zwei Kategorien einteilen: lokales Verhalten und globales Verhalten. Das lokale Verhalten beschreibt die Aktionen und Reaktionen innerhalb eines kleinen, abgegrenzeten Systemteiles. Das globale Verhalten, die auf der Ebene des Gesammtsystems. Wo das lokale Verhalten auch mit heutigen Methoden relativ einfach festzulegen ist, ist das globale verhalten sehr viel schwieriger zu beeinflussen. Neben der höheren Komplexität auf Systemebene, spielt das Phänomen der Emergenz eine wesentliche Rolle. Unter Emergenz versteht man ein Verhalten das auftritt, obwohl es aus der Betrachtung aller zum System gehörigen Komponenten nicht ableitbar ist. In OC Systemen kann solch ein emergentes Verhalten durchaus gewünscht sein, zum Beispiel im bereich der Selbstorganisation. Eine wesentlich wichtigere Aufgabe von Policies ist allerdings die Emmergenz in einem System so zu bescränken, also das globale Verhalten so zu steuern, dass das System seine Aufgabe erfüllen kann.

3 Policyrefinement

Policies ergeben sich aus den Anforderungen an ein System. Betrachtet man wie die Systemanforderungen erarbeitet werden, so stellt man fest, dass es sich um eine kontinuierliche Verfeinerung der Anforderungen handelt, von den wirtschaftlichen, über die fachlichen bis hin zu den technologischen und technischen Aspekten des gewünschten Systems. Für das Erstellen von Policies bedarf es daher auch Methoden und Technologien die diesen Prozess unterstützen. Man

spricht hierbei von Policyrefinement. Der Sinn des Policyrefinement ist analog zu dem des Anforderungsrefinements, aus der Softwareentwicklung, bei dem die Anforderungen an ein System von den fachlichen über die funktionalen bis hin zu den technischen immer genauer ausgearbeitet werden. Eine Policy auf der wirtschaftlichen Ebene, wie zum Beispiel

„Premiumkunden werden bevorzugt behandelt.“

ist bei weitem zu abstrakt, als dass ein Rechner sie verstehen, geschweige denn umsetzen könnte. In weitern Schritten muss genauer geklärt werden, welche Kunden als „Premiumkunde“ gelten und was „bevorzugt behandelt werden“ bedeutet, bis schließlich Regeln entstehen, die vom System umgesetzt werden können. In diesem Beispiel könnten einige davon etwa so lauten:

„Benutze für Premiumkunden den IP-Tunnel 'FASTLANE'“
„Bestellungen von Premiumkunden erhalten den Status 'eilt'“

3.1 Das Policy-Kontinuum

Das Policy-Kontunuum ist ein Modell, dass diese Verfeinerungsschritte in Ebenen(Layers) abbildet, siehe Abbildung 3.1. Der Zentrale Punkt dabei ist, dass die Policies nicht einer Ebene zugeordnet sind, sonder jede Ebene eine Andere Sicht auf die Policies darstellt. Für unser oben genanntes Beispiele bedeutet das, dass: „Premiumkunden werden bevorzugt behandelt.“ die selbe Policy repräsentiert wie: „Benutze für Premiumkunden den IP-Tunnel 'FASTLANE'“ „Bestellungen von Premiumkunden erhalten den Status 'eilt'“. Ersteres ist die Sicht des Business Layers, das Zweite die des Administration Layers. Das bedeutet, dass auf jeder Ebene die selben Policies vorhanden sind, nur in einer unterschiedlichen Repräsentation, und somit dem Bedürfnissen der jeweiligen Ebene angepasst.

Möchte man ein Policy-Kontinuum für ein System implementieren, muss man folgendes festlegen:

- Die verwendeten Abstraktionsebenen.
- Die Policysprache(n) jeder Ebene
- Transformationen für jede Ebene zur darunterliegenden

Im Folgenden sollen die drei verbreitesten Layer des Policy-Kontinuums kurz vorgestellt werden.

3.1.1 Der Business Layer ist eine sehr abstrakte Sicht. Hier werden vor allem wirtschaftliche Aspekte in der Vordergrund gestellt. Da dieser Layer für technisch zumeist nicht geschulte Mitarbeiter gedacht ist, werden die Policies sehr prägnant und intuitiv dargestellt. Es kommen Policysprachen wie SBVR zum Einsatz. Auf Grund des hohen abstraktionsgrades sind die Transformationen meistens nur teilweise automatisierbar.

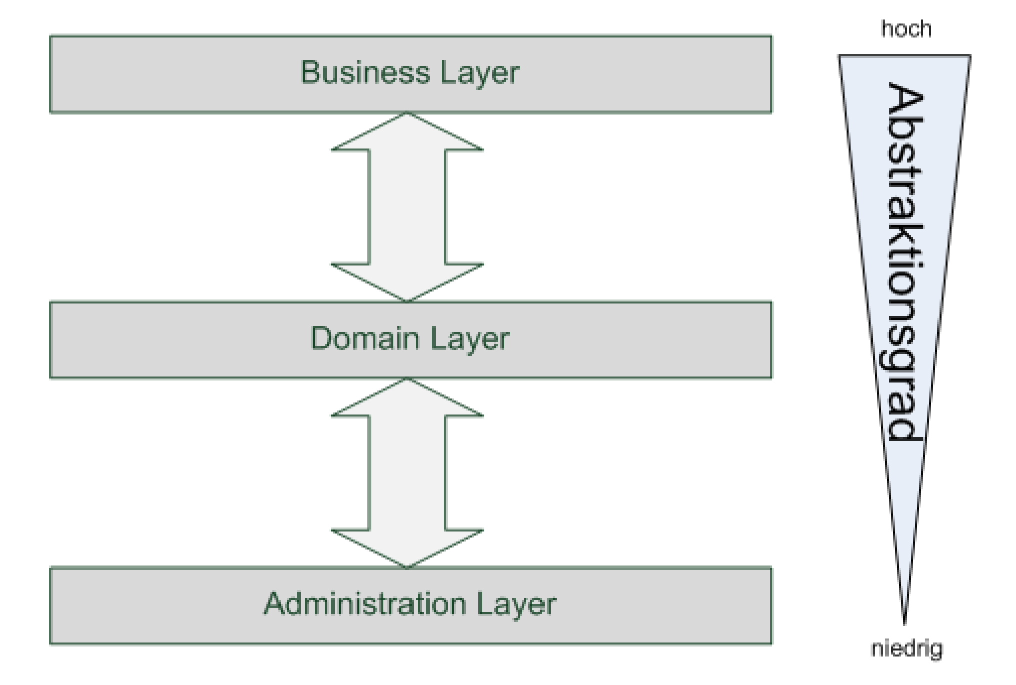


Abbildung 2. Ein Policy-Kontinuum mit den 3 gebräuchlichsten Ebenen

3.1.2 Der Domain Layer

3.1.2 Der Domain Layer ist deutlich konkreter als der Busines Layer, aber immer noch so abstrakt, das die Policies mit dieser Sicht noch nicht von Rechnern verarbeitet werden können. Die Zielgruppe dieser Sicht sind Domänenexpreten, wie Web-, Datenbank- oder CRM-Experten. Die Sprache für diese Sicht müssen sich an die jeweilige Domäne anpassen können, oder es werden domänenspezifische Sprachen verwendet.

3.1.3 Der Administration Layer ist , wie der Name schon sagt, die Sicht für die Systemadministratoren. Die Policies in dieser Sicht sind in einer Form, dass sie dirkt vom System verarbeitet werden, oder von den Administratoren ohne weitere Informationen in eine Konfiguration überführt werden können. Die Policies sind in formalen, für Rechner vearbeitbaren Sprachen, und herstellerspezifischen Formaten notiert.

3.2 Goalbased Policyrefinement

Das in [1] vorgestellte Goal-based Policy Refinement ist eine Technik, die Strategien entwerfen soll, um die Policies in einem System zu erfüllen. Unter Strategie versteht man dabei einen Plan der angibt welche Systemfunktionen in welcher Reihenfolge ausgeführt werden sollen. Der Erste Schritt ist also aus den Policies und den Systemanforderungen die Ziele zu ermitteln. Man erhält dadurch die sogenannten high-level Goals, die noch zu abstrakt sind um mit ihnen weiter zu arbeiten. Für das Refinement der Ziele wird der KAOS-Ansatz von Darimont[2] vorgeschlagen. Das Ergebnis sind Ziele, die so weit verfeinert sind, dass jedes davon von einer einzigen Systemkomponente/-funktion erreicht werden kann. Bei den verfeinerten Zielen weren zwei Arten unterschieden, konjunktive (es müssen alle Unterziele erreicht werden um das übergeordnete Ziel zu erreichen) und disjunktive (es muss nur eines der untergeordneten Ziele erreicht werden um das übergeordnete zu erreichen). Als nächster schritt findet die Zuweisung der low-level ziele zu den Systemkomponenten, die sie erfüllen können statt. Ist das geschehen, kann durch abductive reasoning eine Strategie, also ein Ausführungsplan erstellt werden, bei dem sichergestellt ist, dass das System seine Aufgabe unter Einhaltung aller Policies erfüllt. Für die Umsetzung dieses Refinemntprozesses wird neben den Policies auch eine formale beschreibung des Systems benötigt. Bandara[1] schlägt das Event Kalkül (Event Calculus, EC) vor, da es einerseits formal und damit von Rechnern verarbeitbar ist, sich aber auch für die Modellierung von dynamischen Systemen eignet. Für die eigentliche Modellierung der Policies und des Systems werden allerdings UML Dialekte anstatt des EC verwendet. Vor der Berechnung einer Strategie erden diese UML Modelle zuerst in den EC übersetzt.

3.3 Classificationbased Policyrefinement

Diesem von Hewlett Packard und der North Carolina State University[8] entwickelte Ansatz des Policyrefinement liegt die Annahme zugrunde, dass in high-level Policies Metriken zum Einsatz kommen, die im operativen System nur

als Zusammenwirken von mehreren low-level Metriken existieren. Beispielsweise spricht eine abstrakte Policy von „der Verfügbarkeit einer Datenbank“, im eigentlichen System ist diese aber von vielen Faktoren wie dem Routing im Netzwerk, CPU-Auslastung des Servers, verwendeten Caching Algorithmen, und vielen weiteren abhängig. Zunächst müssen also die verwendeten high-level Metriken in en Policies identifiziert und auf im System messbare low-level Metriken herunter gebrochen werden. Danach führt man auf dem System Testläufe unter realen Bedingungen mit unterschiedlichen Konfigurationen durch. Aus den gesammelten Daten der low-level Metriken wird dann ermittelt unter welchen Bedinungen, das heisst welche Werte die Metriken annehmen dürfen, dass die Policieas eingehalten werden, und wann dies nicht mehr der Fall ist. Da es natürlich ein viel zu großer Aufwand wäre alle Kombinationen für die Konfiguration und Lasten zu testen, benutzt man eine Technik aus dem Bereich des Machine Learning, einen Entscheidungsbaum basierten Classifier. Diese Technik erlaubt es anhand einer kleinen Menge an „Lerndaten“ sehr genaue Vorhersagen über das zu erwartende Verhalten zu machen (98% richtige Vorhersagen vergleiche[8, Table 4])

3.3.1 Classificationbased Policyrefinement Beispiel

Dieses Beispiel ist [8] entnommen. Ziel ist es hier die folgenden high-level QoS Policies auf low-level Metriken herunter zu brechen.

> Antwortzeit von Mo-Fr 08:00-17:00 soll ¡85ms sein.
> Verfügbarkeit von 08:00-20:00 ¿ 99.95
> Compliance über einen Monat ¿ 97
> Es soll weniger als 100 Transaktionen/Sekunde geben.

Nachdem auf einem Testsystem eine Anzahl von Lasttests durchgeführt wurden, erhält man die Testdaten in Form von Tupeln, die wie folg aussehen:

(Metrik 1, Metrik 2, ... Metrik n, TRUE) bzw.
(Metrik 1, Metrik 2, ... Metrik n, FALSE)

Die ersten Einträge sind die gemessenen Werte der low-level Metriken, der letzte gibt an, ob die high-level Policies eingehalten (TRUE) oder verletzt (FALSE) wurden. Anhand dieser Daten kann nun ein Entscheidungsbaum wie in 3 erstellt werden. Betrachtet man diesen Baum, so stellt man fest, dass jeder Pfad von der Wurzel bis zu einem Blatt eine Konstellation bei den Metriken repräsentiert, die zum Einhalten oder einer Verletzung der Policy führen. Da wir daran interessiert sind, dass die Policies eingehalten werden sind für uns die mit TRUE beschrifteten Blätter und ihre Pfade von Interesse. Verfolgt man den Pfad von der Wurzel bis zum Blatt 29 (unten rechts in 3), so kann man daraus diese Policy ableiten:

$(\text{dbCPU} \geq 7.03)\&\&(wwwCPU < 27.2)\&\&(wwwRXPR \geq 8769.72)\&\&dbTXPR < 9666.55)\&\&(RBR \geq 10477.8)\&\&dbTXPR \geq 3790.93)\&\&(wwwWCR \geq 11424.7)$

N ach weiteren Iterationen, bei denen mehrfach auftretende Metriken zusammengefasst werden erhält man folgende Policy:

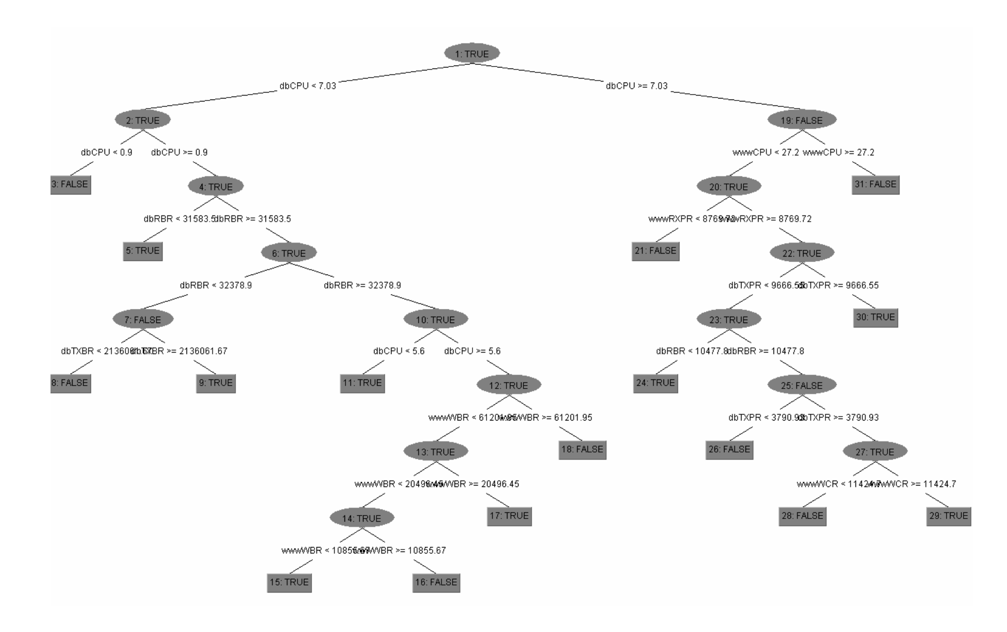


Abbildung 3. Ein mit dem Random Tree Algorithmus berechneter Entscheidungsbaum. Quelle [8]

$(\ 7.03 \leq dbCPU \leq 12.434)\&\&(3.61269 \leq wwwCPU \ll 27.2)\&\&(8769.72 \leq wwwRXPR \leq 183002.0)\&\&(3790.93 \leq dbTXPR < 9666.55)\&\&(10477.8 \leq dbRBR \leq 154804.0)\&\&(11424.7 \leq wwwWCR \leq 12352.3)$

M it dieser Verfeinerten Policy sind dem System nun von ihm beeinflussbare Parameter und ihre Wertebereiche bekannt, um die high-level QoS Policies zu erfüllen.

3.4 Automatisierung der Policytransformationen

Die Automatisierbarkeit der Transformationen zwischen zwei Ebenen nimmt im allgemeinen mit dem Abstraktionsgrad ab. Die in [1] vorgestellte Methode automatisiert beispielsweise alle Schritte von den low-level Zielen, bis hin zur Strategie vollständig, aber die Verfeinerung der high-level Ziele zu low-level Zielen muss manuell durchgeführt werden. Ebenso wie in der Softwareentwicklung können Modelle oft zumindest teilweise transformiert werden. Transformationen auf der selben Ebene sind auf Grund des gleich bleibenden Abstraktionsgrades leichter durch zu führen und daher leuchter zu automatisieren. Als Beispiel soll hier die automatische Transformation der UML Modelle in den Event Kalkül in [1] dienen. Es ist aber durchaus möglich für eine beschränkte Domäne eine vollständige Automatisierung der Policytransformationen zu realisieren. In [7] wird ein Policy Kontinuum für das Management eines Netzwerkes mit 3 Diensten und 3 QoS

Leveln vorgestellt, in dem alle Policies von der obersten bis zur untersten Ebene mit Hilfe von XML und XSLT vollständig automatisch transformiert werden.

4 Policy Konflikte

In jedem System in dem mehr als eine Policy vorhanden ist besteht die Gefahr, dass Policykonflikte auftreten. Die Wahrscheinlichkeit solcher konflikte nimmt noch einmal zu, wenn die Policies von verschiedenen Personen erstellt und gepflegt werden. Zunächst muss geklärt werden welche Arten von Policykonflikten es gibt. In [4] wird die Unterteilung in statische und dynamische Konflikte vorgenommen. Statische Policykonflikte sind unabhängig vom System rein in der Definition der Policies begründet. Das Beispiel in 1 zeigt ein Beispiel bei dem

Tabelle 1. Statischer Policykonflikt

Alle Benutzer der Gruppe Zeichner dürfen auf das Verzeichnis CAD zugreifen.
Der Benutzer TZ3 der Gruppe Zeichner darf nicht auf das Verzeichnis CAD zugreifen.

die erste Policy dem Benutzer TZ3 den Zugriff auf das Verzeichnis CAD erlaubt, und die zweite ihm diesen verbietet. Dynamische Policykonflikte dagegen sind von Systemzustand abhängig, wie das Beispiel in 2 zeigt. Ein Konflikt tritt hier nur auf, wenn sowohl das Flag 'X-PRIORIZE' gesetzt ist, als auch ein Anhang mit mehr als 2MB versendet wird. Eine Spezielle Art der Dynamischen

Tabelle 2. Dynamischer Policykonflikt

Falls das Flag 'X-PRIORIZE' gesetzt ist sende die email über der IP Tunnel 'FASTLANE'
Falls ein Anhang mit mehr als 2MB versendet werden soll benutze nicht den IP Tunnel 'FASTLANE'

Policykonflikte wird in [3] angesprochen, das Problem der Nebenläufigkeit oder Concurrency. Als Beispiel soll hier eine Software zur Fuhrparkverwaltung herangezogen werden, in der eine Policy regelt, dass derjenige der ein Fahrzeug zuerst reserviert es auch bekommt. Wird das selbe Fahrzeug nun aber von zwei verschiedenen Personen gleichzeitig reserviert, kann das System anhand dieser Policy nicht entscheiden was geschehen soll.

Zur auflösung solcher Konflikte gibt es eine Vielzahl an Techniken. Implizite Priorisierung, das Zuweisen expliziter Prioritäten und die Berechnung eise Abstandes zwischen den Policies und dem zu verwaltenden Objektes, sind sind nur einige die hier kurz vorgestellt werden sollen.

4.1 Priorisierung von positiven/negativen Policies

Diese Art der Konfliktlösung ist sehr einfach. Entweder werden negative Policies priorisiert, das heißt wenn etwas von einer Policy verboten wird ist es verboten, auch wenn eine andere Policy es erlaubt, oder es werden positie Policies priorisiert, was bedeutet, dass alles was von einer Policy erlaubt wird auch ausgeführt werden darf. Mit dieser Art der Priorisierung lassen sich jedoch nur solche Konflikte lösen, bei denen Policies das selbe erlauben und verbieten. Betrachtet man das Beispiel in 3 so stellt man fest, dass es keine Einteilung in positive und negative Policy gibt, und der Konflikt daher nicht mit dieser Methode aufgelöst werden kann.

Tabelle 3. Zwei positive Policies

Emails die älter als 2 Monate sind werden archiviert.
Emails die älter als 2 Monate sind und einen Anhang haben werden gelöschet.

4.1.1 Explizite Priorisierung Eine feingranularere Priorisierung lässt sich durch das Zuweisen von expliziten Prioritäten erreichen. Für gewöhnlich sind die Prioritäten Zahlenwerte bei denen ein niedrigerer Wert eine höhere Priorität bedeutet. Der Nachteil hierbei ist, dass die Werte manuell angegeben werden müssen. Gerade bei großen Systemen bei denen mehrere Angestellte für die Pflege der Policies zuständig sind kann das schnell zu Inkonsistenzen bei der Priorisierung führen. Man stelle sich vor dass ein Administrator für die Prioritäten die Werte 1-10 verwendet ein anderer aber die von 1-100.

4.1.2 Abstandsberechnung von Policies Ein Weg die oben genannten Probleme zu umgehen, ist die automatische und objektive Berechnung der Priorität einer Policy. Es wird also nicht mehr für jede Policy ein fester Wert angegeben, sondern eine Funktion, die die Priorität ermittelt. Ein Beispile hierfür ist die Rechteverwaltung in vielen modernen Dateisystemen. Die Prioritätsfunktion ist hier die Ordnerhierarchieebene. Im Normalfall gelten die Zugriffsberechtigungen des übergeordneten Ordners auf Ebene n, gibt es neue Berechtigungen auf Ebene n+m so haben diese eine Höhere Priorität und überschreiben die der Ebene n.

5 Technologien

5.1 SBVR

SBVR (Semantics of Business Vocabulary and Rules) ist eine Spezifikation der OMG. Es spezifiziert eine Sprache zur Beschreibung von Policies auf dem Business Layer der Policy Kontinuums. Um den Anforderungen des Business Layers

gerecht zu werden wurde besonderer Wert darauf gelegt, dass die Policies unabhängig von der IT-Technik (Hardware/Software) und die Syntax für Menshen leicht verständlich und benutzbar ist. Darüber hinaus sollte die Sprache in allen wirtschaftlichen Bereichen einsetzbar sein.
SBVR definiert dafür im wesentlichen zwei Konzepte, Business Vocabulary und Business Rules. Das Business Vocabulary ist eine, aus dem jeweiligen wirtschaftlichen Bereich entliehene Menge von Begriffen und Ausdrücken, zusammen mit einer Eindeutigen Beschreibung ihrer Semantik. Das Konzept der Business Rules definiert eine formale Syntax für Policies. Mit Hilfe des Vokabulars und der Syntax können dann Policies erstellt werden wie refsec:SBVRBeispiel zeigt.
Zusätzlich enthält die SBVR Spezifikation noch Definitionen von XML Schemata für den Austausch von Business Vocabularies und Business Rules.

5.1.1 SBVR Beispiel

5.2 Das Ponder Framework

Das Ponder Framework ist ein Projekt des Imperial College, und mit inzwischen über 15 Jahren Forschun und Entwicklung heute eines der ausgereiftesten [5]. Es bietet neben einer Policy Sprache (Ponder Policy Definition Language) auch eine Sammlung von Management Tools, die im Ponder Toolkit zusammengefasst sind. Die Ponder Policy Specification langeaue ist eine objektorientierte Sprache, die aus anderen objektorientierten Sprachen bekannten Mechanismen der Generalisierung und Instanziierung unterstützt. Weiterhin gibt es die Möglichkeit einfache, so genannte Primitive Policies zu komplexeren Composite Policies zusammenzusetzen. Bei den Primitive Policies wird zwischen fünf Typen unterschieden. Authorisation Policies dienen dazu den Zugriff auf Systemressourcen

Abbildung 4. Primitive Policy Typen bei Ponder

Authorisation Policies
Information Filtering Policies
Delegation Policies
Refrain Policies
Obligation Policies

zu regeln. Sie können diesen entweder gewähren (Positive Authorisation Policy) oder verweigern (Negative Authorisation Policy). Information Filtering Policies transformieren die Ein- und Ausgaben von Systemoperationen. Delegation Policies Regeln die Weitergabe von Authorisation Policies von einem Systemteil an einen Anderen, damit dieser die Berechtigung besitzt eine an ihn übertragene Aufgabe auszuführen. Refrain Policies legen fest, welche Aktionen ein System nicht ausführen soll, obwohl es die Berechtigung dazu besitzt. Der letzte primitive

Policytyp sind die Obligation Policies. Einige der Beispiele in dieser Arbeit sind von diesem Typ (vergleiche Tabelle 2). In [5] werden sie mit „event-triggered condition-action rules“ beschrieben. Es handelt sich also um Policies das die Reaktion des Systems auf ein bestimmtes Ereignis und unter bestimmten Bedingungen regelt. Von den Composite Policies gibt es drei Typen. Diese Typen

Abbildung 5. Composite Policy Typen bei Ponder

Role
Relationship
Management Structure

helfen dabei die Policies ausgehend von den Geschäftsstrukturen zu gliedern, und damit übersichtlicher und leichter wartbar zu machen. Eine Rolle stehen für mehrere Systemteile, die die selbe Aufgabe erledigen, und für die daher die selben Policies gelten. Anstatt nun jedem Dieser Systemteile die Policies einzeln zuzuweisen werden sie in der Rolle gesammelt und jedem Systemteil wird diese Rolle zugewiesen. Relationships sind Rollen sehr ähnlich, allerdings erden hier die Policies zusammengefasst, die die Interaktion von Rollen miteinander regeln. Management Structures sind der abstrakteste Typ der Composite Policies. Sie fassen Rollen und Relationships zusammen, die fin mehreren Unternehmensteilen auftreten. Für eine Universität kann zum Beispiel die Management Structure „Lehrstuhl“ einmal erstellt werden, und muss dann für jeden tatsächlichen Lehrstuhl nur instanziiert werden. Dies setzt natürlich voraus, dass die Rollen und Relationships in jedem Lehrstuhl sehr ähnlich sind.

Zusätzlich zu der Policysprache bietet das Ponder Framework noch einige Tools für das Policy Management. Hier ist vor allem das Compiler Framework, das automatisierte Transformationen von Ponder Policies auf eine Vielzahl von Sicherheitsplatformen, Firewalls, Datenbanken und Betriebssystemen bietet. Ausserdem wird ein Editor für die Policies mit integriertem Compilersupport Angeboten.

5.3 Das IETF/DMTF Policy Framework

Diese Framwork bietet eine Sprache auf Basis der UML. Die Grundlage für alle Policydefinitionen sind das CIM (Core Information Model), das PCIM(Policy Core Information Model) und das QPIM (QoS Policy Information Model). Das CIM definiert einen kleinen Satz von Klssen und Relationen, mit denen man Systeme, Netzwerke, Dienste und Applikationen, also die Umgebung die durch Policies verwaltet werden soll, modelieren kann. Darauf aufsetzend wurde das PCIM entwickelt, das es gestattet Policies zu definieren, und inzwischen durch PCIMe erweitert wurde. Auf dem PCIM setzt schließlich das QPIM auf, dass speziell für QoS Policies im Bereich Netzwerkmanagement bestimmt ist. Auf

Grud des Umfanges dieser Spezifikation (das PCIM alleine enthält bereits 14 Strukturklassen und 15 Assotiazionsklassen, und wurde mit PCIMe nochmals um 30 Struktur- und 27 Assoziationsklassen erweitert siehe [6]) nicht weiter behandelt werden.

6 Aktuelle und zukünftige Forschung

- policy refinement
- distribution der policy in verteilten systemen

Im Fokus der Forschung standen bisher die Policy Sprachen. Im universitären Bereich wurden hier vor allem Sprachen für den Domain Layer, im wirtschaftlichen hersteller- und produktspezifische Sprachen entwickelt. Mit dem IETF/DMTF Framework ist auch schon ein internationaler Standard im Entstehen. Auch für Policytransformationen im Bereich des Domain und Administration Layer ist mit der Ponder Compiler Framework ist bereits ein anwendbares Produkt zu haben.

Nachholbedarf gibt es für die abstrakteren Ebenen des Policy Kontinuum (Business Layer). Es gibt zwar, wie die Entwicklung von SVBR zeigt, auch hier Bemühungen, aber in den Meisten Arbeiten wird der Schritt vom Business Layer zu Domain Layer mehr oder weniger mit nur einem Nebensatz abgehandelt. Ein lohnender Ansatz wäre vermutlich die Überprüfung von bestehenden Mitteln dieser Ebenen (ARIS etc.) auf ihre Anwendbarkeit im Bereich Policy Base Management.

Ein weiteres, noch eher unbeachtetes Thema sind die Einsatzgebiete für PBM. Wie man vor allem an den gewählten Beispielen (auch in dieser Arbeit) sehen kann steht ganz klar das Management von Netzwerken im Rampenlicht. Dies ist verständlich, da PBM besonders für verteilte Systeme, in denen das Netzwerk eine zentrale Rolle spielt, sehr gut geeignet ist. Ob PBM beispielsweise auch für die Konfiguration eines Prozessschedulers auf einem normalen Desktop PC oder einem Server sinnvoll ist, oder sich in diesem Fall hart in das Programm codierte Policies besser eignen wird in keiner im Literaturverzeichnis aufgeführten Arbeit erwähnt.

Literatur

[1] Arosha K. Bandra, Emil C. Lupu, Jonathan Moffett, and Alessandra Russo. A Goal-based Approach to Policy Refinement. In *Proceedings 5th IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2004) IBM TJ Watson Research Centre, New York, USA, June 2004*, June 2004.

[2] R. Darimont and A. van Lamsweerde. Formal refinement pattern for goal-driven requirements elaboration. In *Foundations of Software Engineering*, pages 179–190. 4th Symposium on the Foundations of Software Engineering (FSE4), 1996.

[3] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. A logic programming approach to conflict resolution in policy management.

[4] E. Lupu and M. Solman. Conflict Analysis for Management Policies. In *Proceedings of the 5th International Symposium on Integrated Network Management IM´97 (formerly known as ISINM), San-Diego, USA, Chapman and Hall, May 1997*, May 1997.

[5] P. R. G. of the Imperial College London. The ponder policy based management toolkit. Available online at http://www-dse.doc.ic.ac.uk/Research/policies/ponder/PonderSummary.pdf.

[6] T. poisl. Policy-basiertes management einer it-infrastruktur am beispiel des vpn-dienstes. Diploma Thesis, September 2006. Available online at http://www.atis.uka.de/download/2006_DA_Poisl.pdf.

[7] S. van der Meer, A. Davey, S. Davey, R. Carroll, B. Jennings, and J. Strassner. Autonomic Computing: Prototype Implementation of the Policy Continuum.

[8] Yathiraj B. Udupi, Akhil Sahai, and Sharad Singhal. A classification-based approach to policy refinement.

Modellierung paralleler Abläufe in Organic Computing Systemen

Nikolai Klimov

Universität Augsburg
nikolai1.klimov@informatik.uni-augsburg.de

Zusammenfassung In modernen Rechnern und in verteilten Systemen werden viele Vorgänge parallel ausgeführt mit dem Ziel alle Komponenten gleichmäßig auszulasten oder Leistung zu erhöhen. Diese Vorgänge sind von Natur aus kompliziert. Um sie besser verstehen und analysieren zu können werden verschiedene Modellierungssprachen eingesetzt. Diese Arbeit verschafft einen Überblick über aktuell verwendete Modellierungssprachen, ihre Möglichkeiten und Grenzen.

1 Einleitung

Im Bereich der Betriebssysteme versteht man unter einem Prozess ein auf einem Rechner im Ablauf befindliches Programm, zusammen mit all seinen benötigten Ressourcen [2, 6]. In der Wirtschaft spricht man von Arbeitsabläufen und von Geschäftprozessen, die in einer Organisation/Firma erledigt werden. Der Begriff "Ablauf" und der Begriff "Prozess" werden manchmal als Synomyme verwendet. Anhand eines Beispiels soll der Unterschied zwischen Prozess und Ablauf gezeigt werden, siehe Abb. 1.

Im Kapitel 2 werden Begriffe und Probleme der parallelen Programmierung definiert. Kapitel 3 stellt verschiedene Modellierungssprachen vor und erläutert deren parallele Kontrollstrukturen. Kapitel 4 zeigt anhand eines verteilten Algorithmus, wie man seine Abläufe und Koordination der Prozesse im verteilten System modellieren kann. Am Schluß werden die Vorteile und Nachteile der Modellierungssprachen diskutiert.

2 Grundlagen der parallelen Programmierung

2.1 Nebenläufigkeit und Parallelität

Definition **Nebenläufigkeit** (engl. concurrency)

> "Mehrere Vorgänge, Programme, Objekte oder Prozesse heißen *nebenläufig*, wenn sie voneinander unabhängig bearbeitet werden können" [9, S.433]

Definition **Parallelität**

Ablaeufe

(1) A → B → C → D → E → F

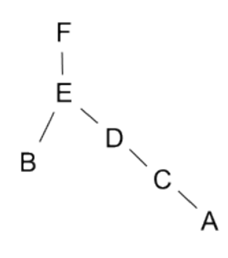
(2) B → A → C → D → E → F

(3) A → • → B → • E → F
  • → C → D → •

(4) B → • E → F
  A → C → D → •

(5) C → D → B → A → E → F

Funktionsbaum



Ablauf mit 2 Prozessen

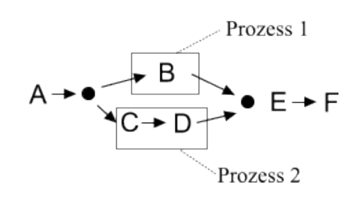


Abbildung 1. Zubereitung von Folienkartoffeln. Die Zubereitung besteht aus 7 Schritten/Operationen/Vorgängen: (A) Karoffeln kaufen (B) Den Backofen auf 250 Grad Celcius vorheizen (C) Die Kartoffeln reinigen und auf ein Stück Folie setzen (D)Folie verschließen (E) Die Kartoffeln auf einem Backbleck im Backofen etwa 50 Min.garen (F) Die Kartoffeln herausnehmen, Alufolie öffnen.
Der Funktionsbaum zeigt die Abhängigkeiten der Vorgänge. Eltern-Knoten sind unmittelbar von ihren Kinderknoten abhängig. Vorgang (C) z.B. benötigt die Daten (Kartoffeln) von (A). Vorgang (F) kann erst am Schluß ausgeführt werden, wenn alle Vorgängerknoten erfolgreich erledigt wurden.
Man muß die Vorgänge in bestimmter Reihenfolge ausführen. Links in der Abbildung sind 5 Abläufe zu sehen. Die Abläufe (1), (2), (3), (4) sind korrekt, Ablauf (4) ist dagegen nicht korrekt.
Wegen der sequentiellen Abfolge ausgeführter Operationenen werden die Abläufe (1) und (2) **sequentieller Prozess** genannt [6, S.588].
Besondere Beachtung verdient der Ablauf (3): Die Zubereitung startet sequentiell mit (A). Wenn (A) beendet ist, starten zwei sequentielle Prozesse {A} und {B,C,D}. Dies bewirkt **Aufspalten** (fork) in zwei unabhängige **Steuerflüsse**. Nachfolgend laufen beide Prozesse voneinander unabhängig ab. Nach einiger Zeit sind alle Operationen in beiden Prozessen erledigt. Dann erfolgt der Übergang vom nebenläufigen zum sequentiellen Ablauf, indem die Steuerflüsse der beiden Prozesse am Synchronisationspunkt (join) vereinigt werden [6, S.589]. Die Stelle wo zwei oder mehr nebenläufige Kontrollflüsse/Pfade zu einem sequentiellen Kontrollfluss zusammengefasst werden nennt man **Synchronisationspunkt**. Am Synchronisationspunkt muß der Ofen die nötige Temperatur erreicht haben und die Kartoffeln vorbereitet sein, bevor der Schritt F beginnen kann.

> "Arbeitsabläufe bzw. deren Einzelschritte heißen *parallel*, wenn sie gleichzeitig und voneinander unabhängig durchgeführt werden können" [9, S.465]

Die Parallelität ist also ein Spezialfall der Nebenläufigkeit. Für zwei nebenläufige Vorgänge A und B ist es ohne Bedeutung, ob zuerst A und dann B oder zuerst B und dann A oder beide gleichzeitig ausgeführt werden. Der Gegensatz zu nebenläufig ist *sequentiell*. Parallele Prozesse laufen unabhängig voneinander ab und interagieren in keiner Weise. Die Ergebnisse der Prozesse hängen nicht von deren Bearbeitung ab.

2.2 Synchronisation

Sehr oft braucht man Prozesse, die miteinander kooperieren, indem sie Zwischenergebnisse und Daten austauschen. Man sagt dass die Prozesse miteinander kommunizieren. Dabei beeinflußen sie gegenseitig ihren weiteren Ablauf, sodass man sie nicht immer in unabhängige Pfade aufspalten kann. Die Bearbeitungsfolge und bestimmte Abläufe müssen eingeschränkt/vermieden werden, da der Zugriff mehrerer Prozesse auf eine gemeinsame Datenstruktur zu nicht korrekten Resultaten bzw. zu Inkosistenzen führen kann.

Bei speichergekoppelten Multiprozessoren befinden sich alle Daten in einem **gemeinsamen Speicher** (shared memory). Die Prozesse können direkt auf alle Daten zugreifen. Der wechselseitige (auch gegenseitiger) Ausschluss (mutual exclusion) garantiert, dass zu einem bestimmten Zeitpunkt nur ein Prozess in einen *kritischen* Bereich eintreten kann [6, S.589][2, S.470]. Der Prozess kann dann alle Datenveränderungen vornehmen, ohne dass Inkostistenzen (race conditions) entstehen. Die Abb. 11 zeigt die Grundidee als Petrinetz modelliert.

Weitere Möglichkeit die Prozesse zu synchronisieren stellt das Prinzip der **Schranke**/Hürde (barrier) dar. Zwei oder mehr Prozesse müssen an einer gewissen Stelle in ihrem Programmcode oder in ihrem Ablauf angekommen sein. Wenn nicht alle Prozesse angekommen sind, können/müssen Prozesse auf die anderen "langsameren" warten. An der Schranke erfolgt der Datenaustausch. Die beschriebene Idee wird auch *Rendevouz* genannt [6, S.590][2, S.469].

Ein weiteres Synchronisationsmuster ist **Nachrichtenübermittlung** (message passing). Ein Prozess A sendet die Nachricht (message) über den Kommunikationskanal, ein anderes sagen wir B, empfängt die Nachricht. Bis zum Eintreffen der Nachricht bleibt B blockiert [6, S.590][2, S.469].

2.3 Verklemmung

In einem Rechnersystem mit mehreren Prozessen können folgende Verklemmungen auftreten: Prozess P1 reserviert Betriebsmittel R1 (z.B.Drucker) und blockiert bis R2 (z.B. Scanner) frei wird. Prozess P2 hat den Scanner schon reserviert und wartet auf Drucker. Die Prozesse verklemmen sich (siehe Abb. 2). Ein ähnlicher Fall kann in einem Transaktionssystem auftreten: Prozess P1 will eine Überweisung von Konto A nach Konto B tätigen und gleichzeitig Prozess

P2 eine Überweisung von Konto B zu Konto A, dann kann es zur Verklemmung kommen, weil jeder Prozess muss Empfänger- und Zielkonto sperren [2, S.474]. Keiner der Prozesse kann weiterarbeiten, sie blockieren sich gegenseitig. Folgende Bedingungen führen zu einer Verklemmung:

- Jede Resource ist entweder verfügbar oder genau einem Prozess zugeordnet.
- Prozesse, die bereits in Besitz von Resourcen sind, können noch weitere Ressourcen anfordern.
- Ringförmige Blockierung von Betriebsmitteln

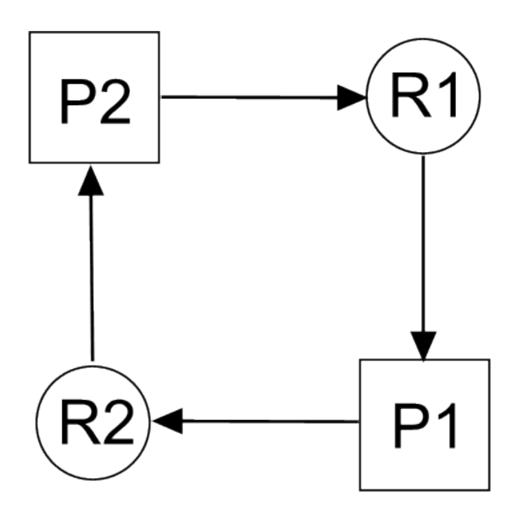


Abbildung 2. Verklemmungssituation. Kreise - Resourcen, Rechtecke - Prozesse. Prozess P1 ist in Besitz der Resource R1 und fordert zusätzlich die Resource R2 an. Prozess P2 ist Besitz von Resource R2 und fordert zusätzlich die Resource R1 an. Keiner der Prozesse kann weiterarbeiten.

2.4 Parallele Kontrollstrukturen

Unter einem Kontrollfluss versteht man in der Informatik die Reihenfolge der Ausführung einzelner Anweisungen, Operationen etc. Um die Richtung des Kontrollflusses grafisch darzustellen werden in den meisten Diagrammenypen durchgezogene Linien mit Pfeilspitze gezeichnet (z.B. in Petrinetzen und in BPMN). Ein Modellierer kann die "Routen" des Kontrollflusses unter anderem mit speziellen Kontrollstrukturen lenken. Jede Kontrollstuktur ist mit einem boolischen Ausdrick ausgestattet, der dann ausgewertet wird wenn der Kontrollfluss an der Kontrollstruktur angekommen ist.

Jede Kontrollstruktur hat immer mindestens einen eingehenden Pfad und mindestens einen ausgehenden Pfad. Man sagt, der Kontrollfluss fließt entlang der Pfade.

Wie am Bespiel in der Abb. 1 gezeigt wurde, kann ein sequentieller Kontrollfluss in zwei oder mehrere nebenläufige Kontrollflüsse aufgespalten werden (fork). Der umgekehrte Vorgang, dass zwei oder mahr nebenläufige Kontrollflüsse vereinigt werden, nennt man join. Diese zwei Arten der Steuerung der Kontrollflüsse können noch in zwei Varianten unterteilt werden.

AND-Split (Parallel Split): erlaubt einen sequentiellen Pfad in zwei oder mehr nebenläufige Pfade aufzuspalten.

AND-Join (Synchronisation): verschmelzt zwei Pfade zu einem. Die Kontrollflüße aller eingehenden Pfade müssen beendet worden sein, bevor der Ablauf fortgesetzt wird.

XOR-Split (Exclusive Choice/Decision): Der Pfad wird in zwei oder mehr alternative Pfade aufgespalten wird. XOR-Split entscheidet welcher an welchem einem Pfad der Kontrollfluß fortgesetzt wird.

XOR-Join (Discriminator) verschmilzt mehrere Pfade zu einem. Wenn der Kontrollfluss von einem Pfad am XOR-Join eintrifft, wird der ausgehende Pfad aktiviert. Später eintreffende Kontrollflüsse von anderen eingehenden Pfaden werden vom XOR-Join ignoriert.

Im nächsten Kapitel wird gezeigt wie diese 4 parallelen Kontrollstrukturen in verschiedenen Modellierungssprachen aussehen.

3 Modellierungssprachen und Techniken

3.1 Bildfahrplan

> "Ein Bildfahrplan (engl. train graph), auch grafischer Fahrplan oder Zeit-Weg-Diagramm genannt, dient der Darstellung der Bewegung von Verkehrsmitteln und tragt zu diesem Zweck die Zeit gegen den Weg auf. Wichtigster Spezialfall ist der Betrieb auf einer Bahn-Strecke, der mit diesem Hilfsmittel grafisch dargestellt wird. Die anderen Fahrplandokumente werden daraus abgeleitet"

[3]. Aus dem Bildfahrplan in der Abbildung 3 lassen sich die Ankunfts-, Abfahrts- und Haltezeiten einzelnen Zügen ablesen.

Die Ordinate repräsentiert die Zeit, die Uhrzeiten verlaufen von oben nach unten. Die Abszisse stellt die Strecke dar, die Haltestellen (Bahnhöfe) werden als vertikale Linien gezeichnet. Die planmäßige Verbindung eines Zuges ist eine farbige zackige Linie, die immer von oben nach unten und entweder von links nach rechts oder von rechts nach links verläuft. Anhand der Linie und der beiden Koordinatenachsen lassen sich Fragen zur Position und Zeit des Zuges beantworten: "Wo befindet sich der Zug zum bestimmten Zeitpunkt?" oder "Wann wird der Zug diesen Ort durchqueren?". Außerdem ist die Geschwindigkeit des Zuges ableitbar: je flacher die Linie desto langsamer und je steiler desto schneller der Zug. Vgl. Abbildung 3

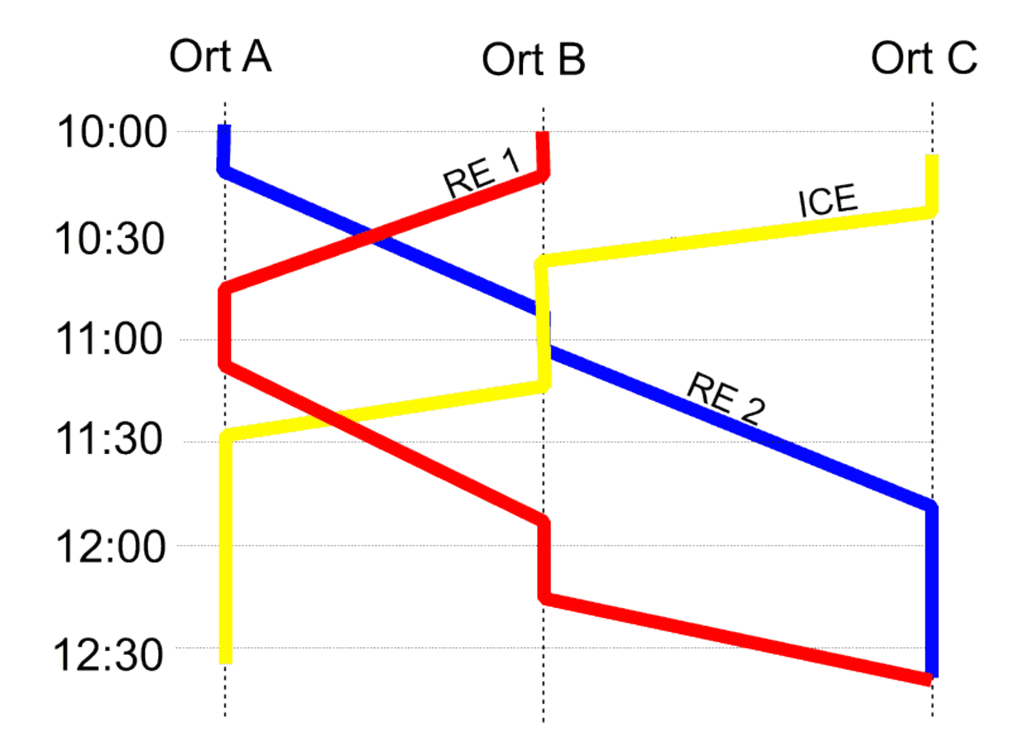


Abbildung 3. Fiktives Bildfahrplan mit drei Zügen RE1, RE2 und ICE und drei Orten bzw. ihren Bahnhöfen. Die Bahn-Strecke zwischen A und B ist zweigleisig, die Strecke B-C dagegen eingleisig ausgebaut.

3.2 Kontrollstrukturen beim Bildfahrplan

Bei der Eisenbahn sind die Gleise eine *kritische Resource* im Sinne der Informatik. Die Züge RE1 und RE2 kreuzen sich auf der Strecke zwischen den Orten A und B um etwa 10:30 Uhr. Angenommen die Strecke A-B ist zweigleisig ausgebaut, dann können die Züge auf verschiedenen Gleisen fahren, eine Kollision ist ausgeschlossen. Gibt es aber nur einen Gleis zwischen den Orten A und B, dann müssen sich zwei Züge ein Gleis teilen und das führt zu einer Kollision. Bei einem Gleis und zwei Zügen, die sich zu einem Zeitpunkt kreuzen, muss ein *gegenseitiger Ausschluß* zwischen den Orten A und B sichergestellt werden, damit es zu keinen Kollisionen kommt.

Jeder Zug fährt unabhängig von anderen Zügen. Die Züge sind vergleichbar damit wie Nachrichten in verteilten Systemen ausgetauscht werden. Prozess A (Bahnhof A) schickt eine Nachricht (Lock mit Wagons) mit der Kennung RE2 an den Prozess B (Bahnhof B). Die Bahnhöfe repräsentieren somit *Synchronisatinspunkte* oder *Prozesse*. Die Züge sind *Nachrichten*. Der ganze Bildfahrplan repräsentiert dann ein *verteiltes System*. Außerdem ließen sich sogar die Passagiere mit dem Inhalt der Nachrichten(Züge) vergleichen. Angenommen am Bahnhof X wartet der Regionalzug RE 2 auf ICE und den Regionalzug RE 1, denn in diesen beiden Zügen fahren z.B. Pendler die unbedingt am Bahnhof X RE2 erwischen müssen, um zur Arbeit zu kommen. Diese Situation ist ähnlich der AND-Join Kontrollstruktur vgl. Abbildung 4. Der AND-Split ist in Abbildung 5 dargestellt: Die ICE-Passagiere haben in Bahnhof X Anschluss zum RE1 und RE2.

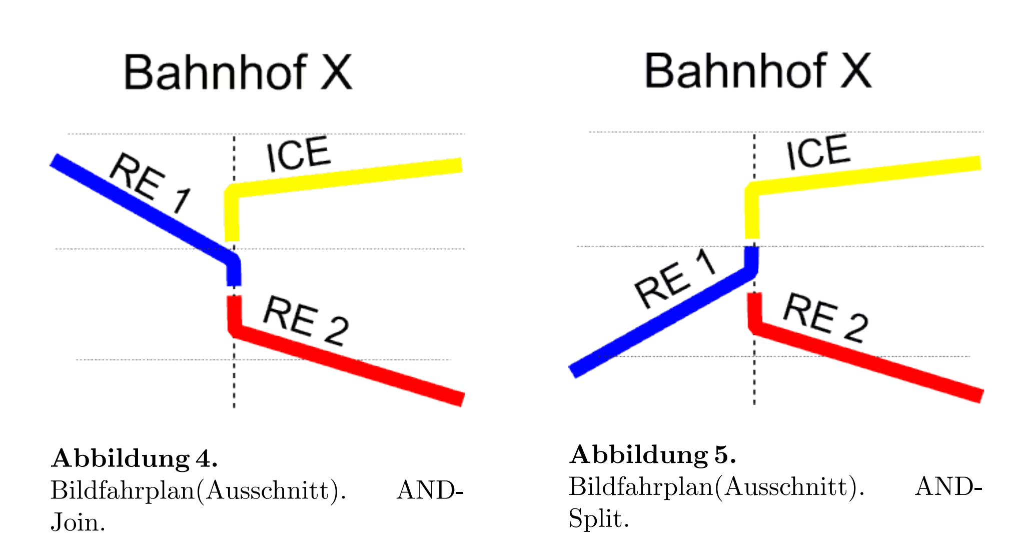


Abbildung 4. Bildfahrplan(Ausschnitt). AND-Join.

Abbildung 5. Bildfahrplan(Ausschnitt). AND-Split.

Heutzutage werden die Bildfahrpläne mit speziellen Programmen erstellt (z.B. OpenTrack), simuliert und auf dem Bildschirm angezeigt, das ermöglicht die Bildfahrpläne mit nützlichen Informationen anzureichern, wie z.B. die Glei-

stopologie, die Anti-Kollisionsüberwachung etc. Der Bildfahrplan stellt eine einfache visuelle Möglichkeit die Züge auf einer Strecke zu überwachen.

3.3 UML-Sequenzdiagramme

Sequenzdiagramme modellieren zeit-basierten Fluß von Ereignissen. Ein Sequenzdiagramm zeigt die teilnehmenden Prozesse oder Objekte oben als Kästchen und die Nachrichten, die zwischen Prozessen ausgetauscht werden als horizontale oder diagonalen Linien, von oben nach unten, in der Reihenfolge in der sie auftreten. Entscheidend ist dabei die Reihenfolge der Ereignisse und nicht der präzise Zeitpunkt, wann ein Ereignis aufgetreten ist. Die Zeit verläuft von oben nach unten. Jedes Objekt hat eine gestrichelte *Lebenslinie*, die vertikal nach unten verläuft. Synchrone Nachrichten werden mit einer gefüllten Pfeilspitze, asynchrone Nachrichten mit einer offenen Pfeilspitze gezeichnet. Die schmalen Rechtecke, die auf den Lebenslinien liegen, sind **Aktivierungsbalken**. Sie zeigen den Focus of Control, das ist der Bereich, in dem ein Objekt über den Kontrollfluss verfügt und aktiv an Interaktionen beteiligt ist. Die Aktivierungsbalken sind aber optional. Das eigentliche Diagramm wird mit einem Rahmen umschlossen. In der linken oberen Ecke des Rahmens befindet sich ein Rechteck, wo *sd* steht, gefolgt vom Diagrammtitel.

Das Sequenzdiagramm in der Abb. 6 modelliert das Drei-Phasen-Handshake-Protokoll, das beim TCP im Internet verwendet wird. Die Sende- und Empfangsereignisse sind durchnummeriert. Als erstes sendet der Client eine SYN-Nachricht an den Server, um zu prüfen, dass der Server sich auf die kommende TCP-Verbindung vorbereitet (Ereignis Nummer 1). Zweitens, wenn der Server SYN-Nachricht erhält (Ereignis Nr.2) und bereit ist eine TCP-Sitzung zu startet, sendet er eine SYN+ACK(SYN)-Nachricht zurück zum Client (Ereignis Nr.3). Wenn der Client SYN+ACK-Nachricht erhält (Ereignis Nr. 4) weiß er, dass seine erste SYN-Nachricht den Server erreicht hat. Schließlich sendet der Client ACK(SYN)-Nachricht, um dem Server zu bestätigen, dass seine SYN+ACK(SYN)-Nachricht angekommen war. Das Drei-Phasen-Handshake-Protokoll wird bei der Initial-Phase von TCP verwendet, damit sich beide Seiten über die Folgenummerierung der Pakete einigen können (weitere Informationen finden Sie in [7]).

Interaktionen können sehr komplex werden, deswegen hat UML2 **kombinierte Fragmente** eingeführt. Ein kombiniertes Fragment besteht aus einem *Interaktionsoperator* und einem oder mehreren *Interaktionsoparanden.* Im Sequenzdiagramm in der Abb. 8 ist ein Alternatives Fragment zu sehen. Der Interaktionsoperator heißt **alt**, die 2 Interaktionsoparanden sind **condition** und **else**. Jeder Operand enthält Interaktionsframente, d.h. Teile des Sequenzdiagramms, die alternativ ausgeführt werden. Der Operand dessen boolischer Ausdruck (guard expression) zu "True" ausgewertet wird, wird gewählt (weitere Informationen zu kombinierten Fragmenten finden Sie in [5, Seite 468]).

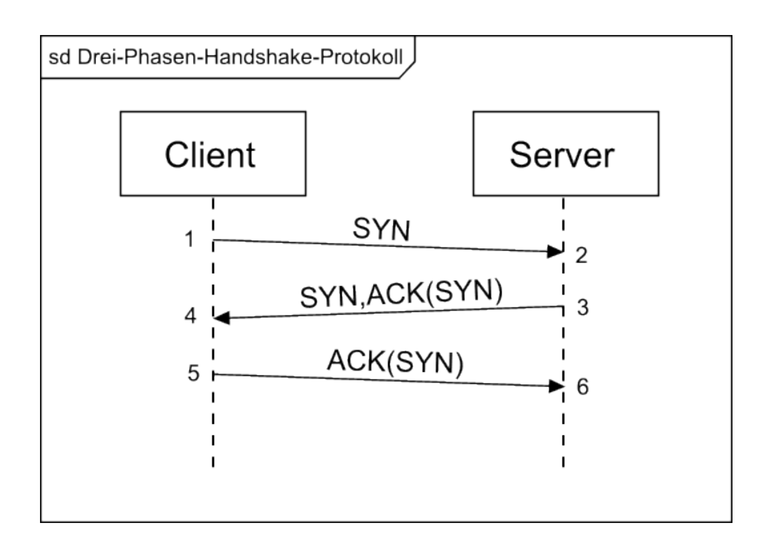


Abbildung 6. Beispiel für ein Sequenzdiagramm. Das Diagramm modelliert Drei-Phasen-Handshake-Protokoll. Die horizontalen Nachrichten-Linien sind mit Nachrichtennamen beschriftet. Die Ereignisse sind durchdummeriert von 1 bis 6: Ereignis 1 tritt von Ereignis 2 auf, Ereigniss 2 vor 3 uns so weiter.

3.4 Kontrollstrukturen bei Sequenzdiagrammen

Die Aktivierungsbalken bei Sequenzdiagrammen sind optional, deswegen gibt es keine Möglichkeit den Start, das Beenden oder das Anhalten von Operationen grafisch darzustellen. Deswegen ist textuelle Beschreibung nötig. Die Prozesse in den Abbildungen 7 8 9 und 10 synchronisieren sich durch Nachrichtenübermittlung, wie im Kapitel 2 beschrieben. Die ausgehenden Nachrichten stoppen die aktive Ausführung des sendenden Prozesses. Der Prozess blockiert, bis er neue Nachrichten empfängt. Die Nachrichten selbst können nützliche Informationen speichern, z.B. Daten an denen die Prozesse arbeiten.

AND-Split: In der Abb. 7 wartet (bzw. blockiert) Prozess X und Prozess Z solange mit der Ausführung bis sie eine Nachricht von Y erhalten. Wenn der Prozess Y beide Nachrichten gesendet hat, blockt er, d.h. er stoppt mit der Ausführung, weil er die Kontrolle an X und Z übergeben hat. X und Z werden nach dem Erhalt der Nachricht von Y anfangen nebenläufig zu arbeiten.

XOR-Split: Im Sequenzdiagramm in der Abb. 8 übergibt der Prozess Y die Kontrolle entweder an X oder Z. Das hängt davon ab, wie die Bedingung *condition* ausgewertet wird. Wenn "True" dann sendet Y an Z, andernfalls sendet Y an X. Um solche Entscheidungen grafisch darzustellen wird **Alternatives Fragment** benötigt.

AND-Join: Normalerweise ist die Reihefolge der Ereignisse bei den Sequendiagrammen wichtig. Mit **Parallem Fragment** kann man nebenläufige Teile der Interaktion modellieren. In der Abbildung 9 senden X und Z an Y. Es spielt keine Rolle ob zuerst die Nachricht von X bei Y eintrifft und dann die Nachricht

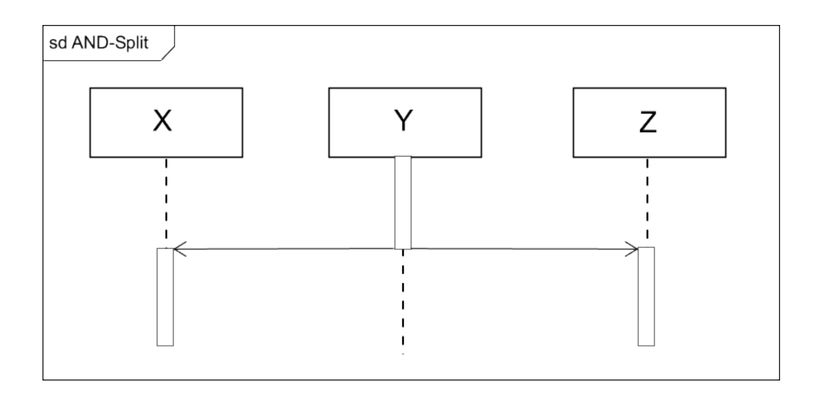


Abbildung 7. Sequenzdiagramm. AND-Split

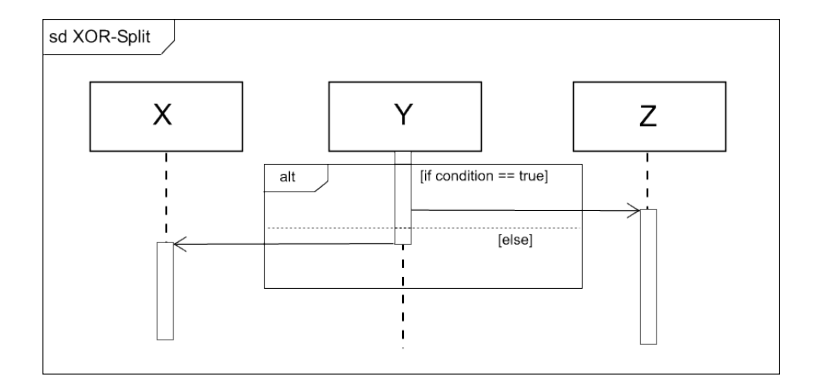


Abbildung 8. Sequenzdiagramm. XOR-Split

von Z oder umgekehrt. Nachdem Y *beide* Nachrichten erhalten hat, wird Y die aktive Ausführung starten.

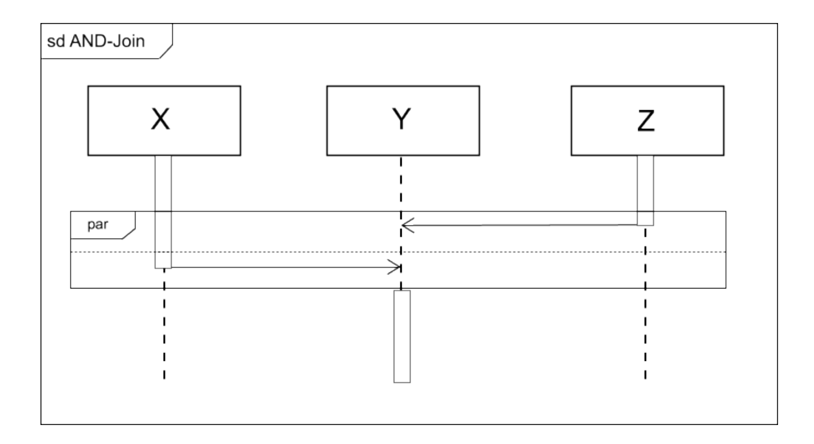


Abbildung 9. Sequenzdiagramm. AND-Join

XOR-Join: Der Prozess Y in der Abb. 10 startet die aktive Ausführung sobald er die Nachricht entweder vom Prozess X oder vo Prozess Z erhält. Angenommen die Nachricht vom Z kommt bei Y als erset an. Y startet daraufhin seine aktive Ausführung. Später eintreffende Nachricht vom X wird vom Y ignoriert.

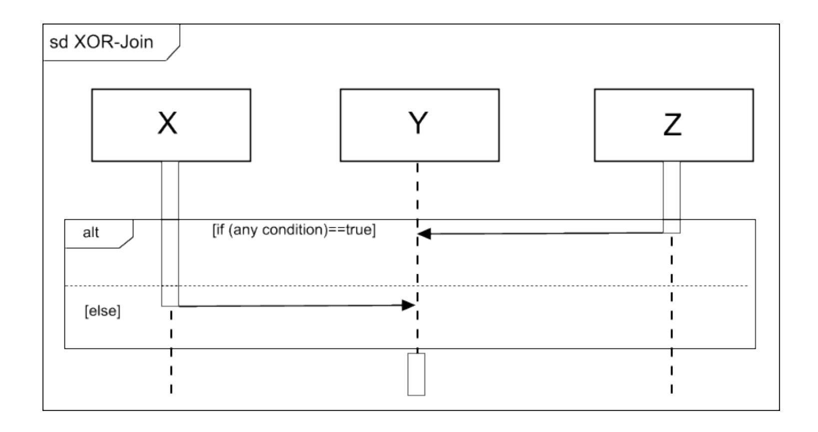


Abbildung 10. Sequenzdiagramm. XOR-Join

3.5 Petrinetze

Petrinetz (engl. petri net): "Modell zur Beschreibung und Analyse von Abläufen mit nebenläufigen Prozessen und nichtdeterministischen Vorgängen" [9, Seite 487]

Petrinetze beschreiben dynamische Systeme, mit einer festen Grundstruktur (z.B. Betriebssysteme, Büroabläufe oder Organisationsabläufe bei Herstellungsverfahren). Ein Petrinetz ist ein gerichteter Graph, der aus zwei verschiedenen Sorten von Knoten besteht: **Stellen** und **Transitionen**. Stellen (auch Plätze genannt) können folgende Rollen im Petrinetz spielen:

- Kommunikationsmedium, z.B. Telefonlinie, Kommunikationsnetzwerk
- Buffer, z.B. ein Lagerhaus, eine Warteschlange oder ein Briefkasten
- Geographische Position, z.B. ein Platz in einem Buro oder Fabrik

Rolle der Transitionen:

- Ereignis, z.B. Starten einer Operation, Umschalten von Geräten wie Licht ein, Licht aus
- Transport eines Objekts, z.B. Warentransport
- Senden von Daten
- Transformation an einem Objekt, wie Aktualisieren einer Datenbank

Stellen werden durch Kreise, Transitionen durch Rechtecke oder durch Balken dargestellt. Die **Kanten** (= Verbindungen zwischen den Knoten) dürfen nur zwei verschiedene Sorten von Knoten verbinden. Die Stellen, von denen die Kanten zu einer Transition laufen, heißen Vorbereich von t, und die Stellen, zu denen von t aus Kanten führen, heißen Nachbereich (oder Output-Stellen) von t. In der Abb. 11 ist Petrinetz zu sehen, das aus 9 Knoten besteht: s1,s2,s3,s4,,s5,t1,t2,t3,t4 . Die Kanten sind durch Pfeile dargestellt. Transition t1 hat im Vorbereich zwei Stellen s1 uns s4 und Nachbereich eine Stelle s3. Transition t3 hat im Vorbereich eine Stelle s3 und im Nachbereich 2 Stellen s1 und s4.

Ein Ablauffluß wird mit Hilfe von **Marken** (tokens) simuliert. Marken können in Petrinetzen folgende Rollen spielen:

- ein physisches Objekt, z.B. ein Produkt, ein Teil, eine Ware
- Information, z.B. eine Nachricht, ein Signal
- Menge von Objekten, z.B. Lieferwagen mit Waren, eine Datei
- Zustandsindikator, z.B. der Zustand in welchem sich der Prozess befindet
- Bedingungsindikator, z.B. Vorhandensein von einem Token an einer Stelle sagt aus, dass eine bestimmte Bedingung erfüllt ist

Die Marken haben bestimmten Datentyp. Verwendet man den Datentyp *bool*, dann heißt das Petrinetz auch "Bedingungs-Ereignis-System". Die Stellen dürfen 0 oder mehr Marken enthalten. Eine Transition ist dann aktiviert, wenn jede der eingehenden Stellen mindestens ein Marke enthält. Man sagt eine aktivierte Transition kann "feuern" bzw. schalten. Wenn eine Stelle feuert, konsumiert sie eine Marke von jeder Stelle im Vorbereich und produziert eine Marke fur

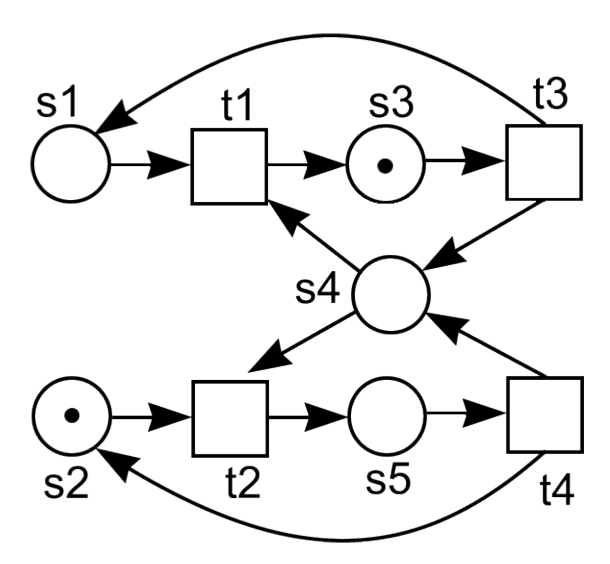


Abbildung 11. Petrinetz. Dieses Petrinetz modelliert wechselseitigen Auschluss zwischen zwei unendlich laufenden nebenläufigen Prozessen. Prozess A befindet sich im kritischen Bereich. Prozess B kann in den kritischen Bereich nicht eintreten, da Stelle s4 (kritische Ressource) leer ist und deswegen Transition t2 nicht schalten kann. Leere Stelle s4 entspricht der Situation, dass einer der Prozesse sich im kritischen Bereich befindet. . Erster Prozess hat die Knotenmenge A={s1,t1,s3,t3}. Zweiter Prozess hat die Knotenmange B={s2,t2,s5,t4}. Die Stelle s4 spielt dabei dir Rolle des Synchronisationspunktes. Das Netz ist in der Anfangsmarkierung. Befindet sich eine Marke in s4 heißt das, dass der kritische Bereich frei ist und einer der Prozesse kann in den kritischen Bereich eintreten, indem die Marke aus s4 entnommen wird. Liegt gleichzeitig in s1,s2 und s4 je eine Marke, dann kann einer der Prozesse sich entscheiden in den kritischen Bereich einzutreten, indem die Transition t1 oder t2 schaltet. Wenn t1 schaltet, entnimmt sie je eine Marke aus s1 und s4 und legt eine Marke in die Stelle s3. Wenn t2 schaltet entnimmt sie Marken aus s2 und s4 und legt eine Marke auf s5. Die Invariante für dieses Netz lautet: die Stellen s3 und s5 dürfen niemals eine Marke gleichzeitig haben oder anders ausgedrückt, die Summe der Marken von s3 und s5 ist gleich 1.

jede Output-Stelle (Nachbereich). Feuern ist atomar, d.h. der Vorgang des Wegnehmens der Marken aus den Input-Stellen der feuernden Transition und des Produzierens des Marken fur die Output-Stellen darf nicht unterbrochen werden. Mehrere Transitionen können aktiviert sein, welche Stelle zuest feuert ist nichtdeterministisch. Der Zustand des Netzes ist repräsentiert durch die Verteilung der Marken uber die Stellen(ganannt auch als **Markierung**).

3.6 Kontrollstrukturen bei Petrinetzen

AND-Split: Wenn Transition t1 in der Abb. 12 schaltet, konsumiert sie die Marke in s1 und produziert je eine in s2 und s3. Der Prozess mit der Knotenmenge {s2,t2} und Prozess mit der Knotenmenge {s3,t3} werden nebenläufig ausgeführt.

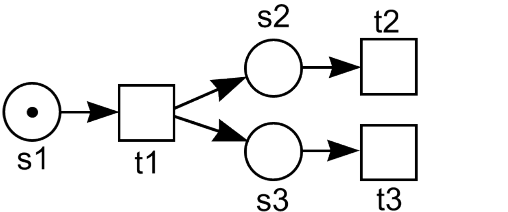


Abbildung 12. Petrinetz. AND-Split.

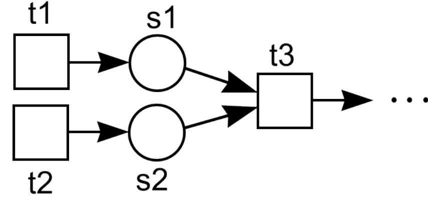


Abbildung 13. Petrinetz. AND-Join bzw. Synchronisation

XOR-Split: Beide Transitionen t1 und t2 in der Abbildung 14 sind aktiviert und können schalten. Die Transition, die zuerst schaltet entfernt die Marke aus s1 und fügt eine Marke in ihren Nachbbereich hinzu (im Bild mit drei Punkten angedeutet). Die andere Transition, die nicht geschaltet hat kann nicht schalten und damit auch nicht eine Marke in ihren Nachreich legen. Somit wird nur derjenige Pfad des Ablaufs gewählt, der die Transition t1 als Knoten hat. Welche Transition zuerst schaltet ist nichtdeterministisch.

AND-Join: Transitionen t1 und t2 in der Abb. 13 können schalten, weil ihr Vorbereich leer ist. Transition t3 kann nur dann schalten, wenn in beiden Stellen s1 und s2 mindestens eine Marke liegt. Wenn t3 schaltet entfernt sie je eine Marke aus den Stellen s1 und s2 und legt eine Marke in ihren Nachbereich (im Bild mit drei Punkten angedeutet). Der Ablauf mit Transition t3 kann also dann ausgeführt werden, wenn beide nebenläufige Prozesse {t1,s1} und {t2,s2} mindestens einmal geschaltet haben.

XOR-Join: Beide Transitionen t1 und t2 in der Abb. 15 können schalten, t3 aber nicht. Schaltet t1 oder t2, liegt eine Marke in s1, Transition t3 ist dann aktiviert und könnte schalten. Es ist also nicht erforderlich, dass t1 und t2 schalten, damit der Ablauf ab t3 fortgesetzt wird.

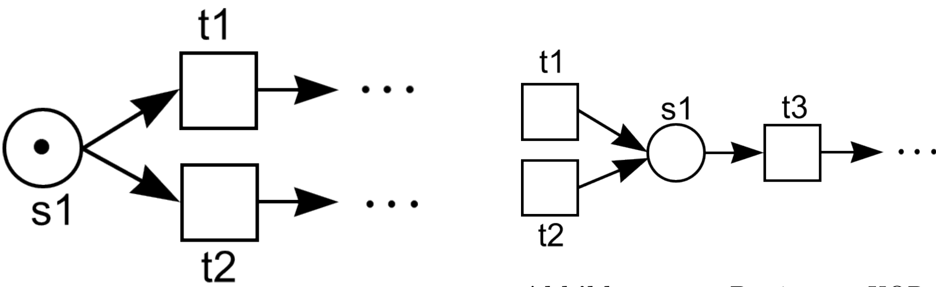


Abbildung 14. Petrinetz. XOR-Split bzw. Alternative.

Abbildung 15. Petrinetz. XOR-Join

3.7 Business Process Modeling Notation

> Business Process Modeling Notation (BPMN) stellt eine Notation zur Verfügung, die leicht verständlich für alle Geschäftspartner ist. Die Wirtschaftsanalysten können damit ihre Anfangsentwürfe für Prozesse erstellen. Die Entwickler sind verantwortlich für die Implementierung mithilfe der Technnologien, welche diese Prozesse ausführen werden. Schließlich überwachen und verwalten die Geschäftsleute die Prozesse. Die BPMN schafft somit einen Standard wie für das Design von Geschäftsprozessen, so auch für die Prozessimplementierung. [4, Seite 1]

Ein Business Process Diagram (BPD) besteht aus einer Menge grafischer Elemente, die in 4 Hauptkategorien eingeteilt sind:

- Flow Objects
- Connecting Objects
- Swimlanes
- Artifacts

Flow Objects sind die wichtigsten grafischen Elemente, um das Verhalten von Geschäftsprozessen zu definieren. Es gibt 3 Flow Objects:

- Events
- Activities
- Gateways

Es gibt 3 Möglichkeiten Flow Objects miteinander zu verbinden oder andere Informationen mit Flow Objects zu verbinden:

- Sequence Flow
- Message Flow
- Association

Es gibt 2 Möglichkeiten die wichtigsten grafischen Elemente zu gruppieren:

- Pools

– Lanes

Artifacts werden verwendet, um zusätzliche Informationen bereitzustellen. Es gibt drei Artifacts, aber Designer oder Modellier-Werkzeuge dürfen zusätzliche Artefakte einfügen. Zur Zeit sind 3 Artefakte standardisiert:

– Data Objects
– Group
– Annotation

Abbildung 16 zeigt nur diejenigen Elemente, die im Kapitel 4 bei Business Process Diagrammen verwendet wurden.

BPMN unterscheidet 3 Typen von Prozessen: Process, Sub-Process und Task. Jeder Typ wird als abgerundeter Rechteck dargestellt. Der Grund für verschiedene Namen reflektiert die hierarchische Beziehungen unter ihnen. Ein Business Process Diagramm modelliert immer einen Prozess, der normalerweise aus beliebig vielen Unterprozessen besteht. Eine Task darf nicht weiter unterteilt (decomposed) werden. Eine **Aktivität** ist ein allgemeiner Begriff für Arbeit, dass eine Organisation leistet. Task und Sub-Process sind Aktivitäten. Ein **Event** ist etwas was "passiert". Sie beeinflüßen den Fluß des Prozesses. Es gibt 3 Grundtypen von Events: Start- Intermediate- und End Events. Jeder Grundtyp hat weitere Typen, die nach ihren Triggern unterschieden werden. Events sind Kreise, im Zetrum können Marken platziert werden, um verschiedene Trigger zu unterscheiden. Verschiedene Linienstiele dienen zur Unterscheidung von Event-Grundtypen. Start Events haben eine Linie, Intermediate Events zwei Linien, End Events eine dicke Linie (siehe Abb. 16). Der Begriff "Ereignis" ist allgemein genug, um viele Dinge im Geschäftprozess zu beschreiben: Starten einer Aktivität, Beenden einer Aktivität, Zustandswechsel eines Dokumentes, eine Nachricht, die eintrifft, etc, all das könnte man als Ereignisse auffassen. Start- und die meisten Intermediate Events haben "Auslöser" (Trigger), die die Ursache des Ereignisses beschreiben. Es gibt 6 Trigger von Start Events in BPMN: **None**, Message, Timer, Rule, Link und Multiple. None-Start-Event wird in beiden Business Process Diagrammen im Kapitel "Machbarkeitsstudie" verwendet. Start Events *starten* eine Aktivität, der Trigger bestimmt welches Ereigniss das Event auslöst. Es gibt 8 Trigger von **Intermediate Events**: None, **Message**, Timer, Error, Compensation, Cancel, Rule, **Link** und Multiple. Diese Event-Typen zeigen verschiedene Möglichkeiten, wie eine Aktivität *unterbrochen werden kann nachdem sie gestartet ist.* Jeder Event-Typ besitzt ein grafisches Symbol, das in der Mitte des Event-Kreises platziert wird.

Ein **Gateway** wird verwendet um das Aufspalten und Zusammenführen von sequentiellem Fluß (Sequence Flow) zu kontrollieren. Gateways werden als Rauten dargestellt. Interne Symbole zeigen die Art der Gateways (z.B. Parallel gateway mit "+" in Abb. 17 oder XOR gateway mit leerer Raute in Abb. 20).

Sequence Flow zeigt die Reihenfolge in welcher Aktivitäten in einem Prozess abgearbeitet werden.

Sequence Flow

Message Flow

Association

Task 1 Activity

Exclusive Gateway

start event: None

end event: None

intermediate event: Message

intermediate event: Rule

intermediate event: Link (throwing)

intermediate event: Link (catching)

Text Annotation Text Annotation

Nachricht
[descrition] Data Object

Buyer Task 1 Task 2 Pool

Abbildung 16. Einige grafische Elemente der BPMN

Message Flow zeigt den Fluß der Nachrichten zwischen zwei Teilnehmern, die bereit sind zu senden oder zu empfangen. Zwei getrennte Pools im Diagramm repräsentieren je einen Teilnehmer.

Eine **Association** verknüpft Informationen mit Flow Objects.

Ein **Pool** repräsentiert einen Teilnehmer.

Data Objects sind Artefakte, weil sie keinen direkten Einfluß auf Sequence Flow or Message Flow haben, aber sie bieten Informationen über das was Aktivitäten für die Arbeit benötigen oder/und was sie produzieren werden.

Text Annotations ist ein Konstrukt für die Modellierer, um zusätzliche Informationen für die Leser eines BPMN Diagramms zur Verfügung zu stellen.

3.8 Kontrollstrukturen bei Business Process Diagrammen

Kontrollstrukturen bei Business Process Diagrammen
AND-Split: BPMN benutzt den Begriff "Gabel" um das Konzept der Teilung eines Pfades in zwei oder mehr parallele Pfade zu bezeichnen. Wenn Task1 in der Abb. 17 beendet wird und teilt der Gateway den Kontrollfluß in zwei Pfade. Task2 und Task3 werden nebenläufig ausgeführt. In BPMN verwendet man **Parallel Forking gateway**, um AND-Split auszudrücken. Der Raute hat ein "+" innen drin. **XOR-Split**: **Exclusive Decision gateway** in der Abb. 18

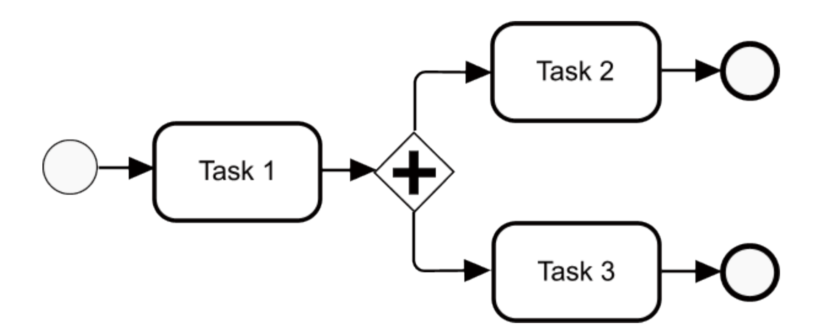


Abbildung 17. BPMN. AND-Split

lenkt den Kontrollfluss in Abhängigkeit vom Wert der boolischen Ausdrücke (condition1 und condition2). XOR gateway wählt den Pfad, dessen boolischer Ausdruck "True" ist. Die Raute bleibt entweder leer oder es wird ein "X" platziert. **XOR-Join**: BPMN benutzt den Begriff "Zusammenfügen" um exklusives Zusammenführen von zwei oder mehr Pfaden zu einem zu bezeichnen. In der Abbildung 20 fließt der Kontrollfluss weiter, sobald einer der beiden nebenläufigen Tasks (Task2 oder Task3) beendet wurde. Kontrollflüsse von später beendeten Tasks werden am **Exlusive Merge gateway** abgewiesen/verworfen. **AND-Join**: BPMN benutzt den Begriff "Vereinigen" um das Zusammenführen von zwei oder mehr parallelen Pfaden zu einem zu bezeichnen (auch bekannt als

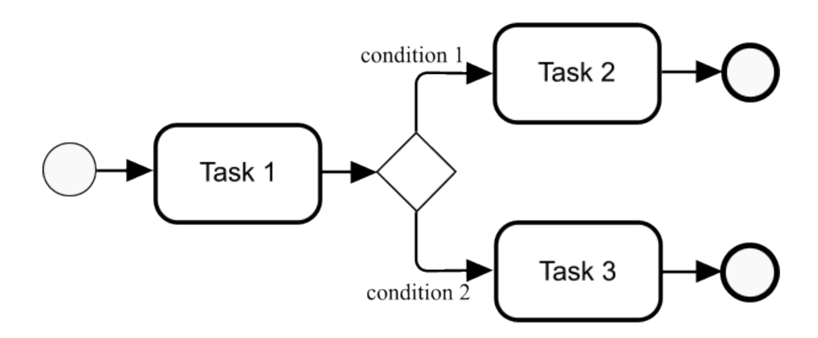


Abbildung 18. BPMN. XOR-Split

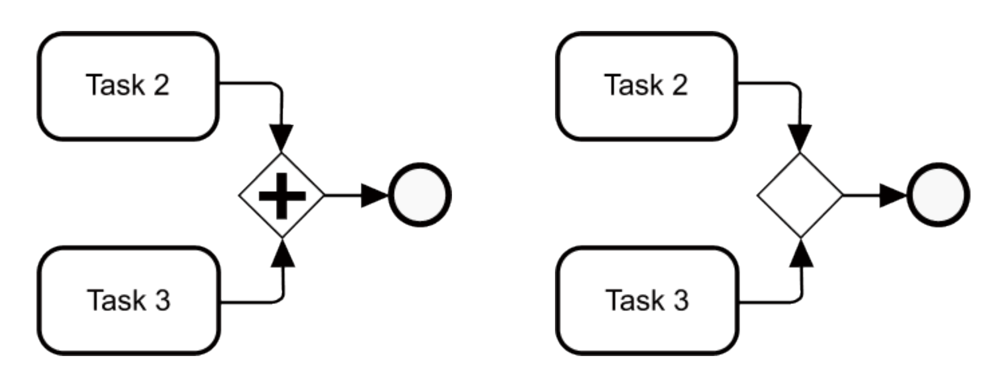


Abbildung 19. BPMN. AND-Join **Abbildung 20.** BPMN. XOR-Join

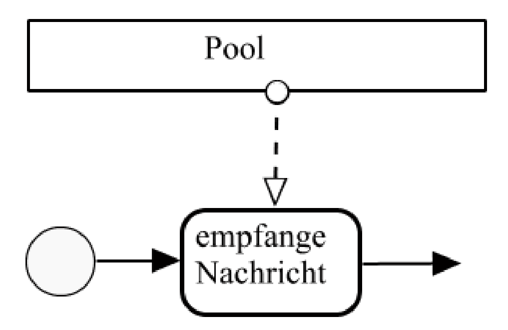


Abbildung 21. BPMN (Ausschnitt). AND-Join. Start-Event startet irgendwann die Task "Empfange Nachricht". Die Task blockiert solange bis eine neue Nachricht vom Kommunikationspartner eintrifft, d.h. erst eine Nachricht kann diese Task beenden.

Synchronisation). In der Abbildung 19 wird am **Parallel Joining gateway** solange "gewartet", bis Task2 und Task3 fertig sind, d.h. die Flüße von allen eingehenden Pfaden müssen beendet werden, damit der Gateway den weiteren Fluß erlaubt. In der Abbildung 21 ist eine weitere Möglichkeit für AND-Join gezeigt. Der Synchronisationspunkt ist kein Gateway, sondernt eine Task, die auf eine Nachricht vom Kommunikationspartner wartet.

4 Machbarkeitsstudie

In diesem Kapitel soll ein verteilter Algorithmus [8, S.304] mithilfe von Sequenzdiagrammen, Petrinetzen und Business Process Diagrammen modelliert werden. Der Algorithmus löst folgendes Problem: wenn zwei oder mehr verteilte Prozesse gleichzeitig gemeinsam genutzte Datenstukturen lesen oder aktualisieren, kann es zu Inkostinzenzen führen. Damit dass nicht passiert muss ein wechselseitiger Ausschluss sichergestellt werden. Der Algorithmus[1] funktioniert wie folgt. Jeder Prozess kann einen von drei Zuständen annehmen:

1. Zustand "Ruhe": Prozess befindet sich nicht im kritischen Bereich und will auch nicht in diesen eintreten.
2. Zustand "Angefordert": Prozess will in den kritischen Bereich eintreten.
3. Zustand "Kritisch": Prozess befindet sich im kritischen Bereich, er liest oder aktualisiert irgendwelche Daten.

Es gibt zwei Arten von Nachrichten: Anforderungsnachricht (ID,Zeit) und Bestätigungsnachricht (OK-Nachricht). Jeder Prozess im System verhält sich nach folgenden Regeln in Abhängigkeit davon in welchem Zustand er sich befindet:

- Zustand "Ruhe": Wenn neue Nachricht eintrifft, egal welcer Art, sendet der Prozess eine OK-Nachricht an den Sender.
- Zustand "Angefordert ": Der Prozess will in den kritischen Bereich[2] eintreten. Dazu erzeugt er eine Anforderungsnachricht mit Prozess-ID und aktueller Zeit (Zeitstempel) und verschickt diese an alle anderen Prozesse im System. Nach dem Senden geht der Prozess in den Zustand "Angefordert" über und wartet solange bis er von allen anderen Prozessen Besätigungen(OK-Nachricht) erhält. Es kann aber sein, dass andere Prozesse zur gleichen Zeit auch in den kritischen Bereich eintreten wollen. Welcher Prozess den Vorrang hat entscheidet der Zeitstempel. Der Niedrigere gewinnt. Wenn neue Anforderungensnachricht eintrifft, vergleicht der Prozess den Zeitstempel aus der empfangenen Nachricht mit dem Zeitstempel aus der eigenen gesendeten Nachricht. Wenn der Empfangszeitstempel kleiner als eigener Zeitstempel, sendet der Prozess OK-Nachricht an den Sender. Wenn aber umgekehrt,

[1] Zwei Voraussetzungen müssen allerdingt erfüllt sein, damit der Algorithmus richtig funktioniert: erstens alle Ereignisse im System sind vollständig sortiert und zweitens alle Nachrichten werden zuverlässig gesendet d.h. ohne Verluste.

[2] Im verteilten System kann es mehrere kritische Bereiche geben. In unserem Beispiel gehen wir davon aus, dass es nur einen kritischen Bereich gibt.

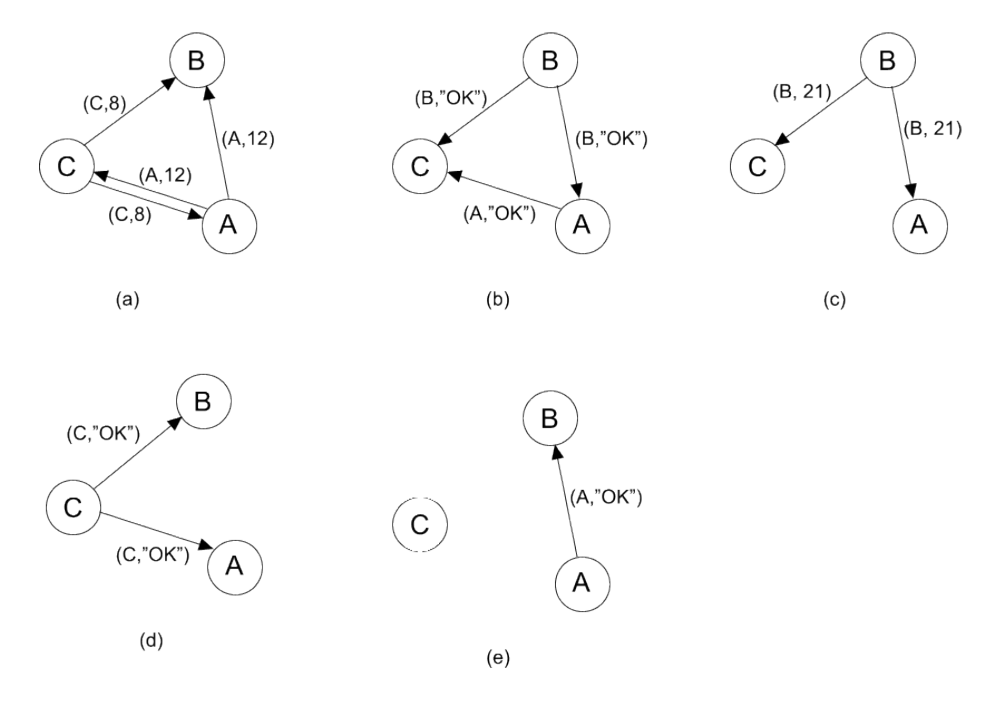


Abbildung 22. (a) Die Prozesse A und C entschließen sich unabhängig voneinander zur gleichen Zeit in den kritischen Bereich einzutreten. Dazu sendet jeder eine Anforderungsnachricht an alle anderen im System. Anforderungsnachricht besteht aus Prozessidentifikation und der Sendezeit.
(b) Da sich B im Zustand "Ruhe" befindet, sendet er Bestätigungsnachrichten an A und C. Als A eine Anforderungsnachricht von C erhält, befindet er sich im Zustand "Angefordert". Deswegen vergleicht Prozess A den empfangenen C-Zeitstempel mit seinem eigenem Zeitstemel. C-Zeitstemel gewinnt, wegen 8 kleiner 12. A hat "verloren" und sendet deswegen eine Bestätigungsnachricht an C. C seinerseits macht den gleichen Vergleich und stellt fest, dass er gewonnen hat, deswegen stellt C A-Anforderung in seine Warteschlange. C hat nun von allen Prozessen im System eine Bestätigung erhalten und tritt in den kritischen Bereich ein.
(c) B will in den kritischen Bereich eintreten und sendet deswegen Anforderungen an C und A. C befindet sich im kritischen Bereich, deswegen stellt C B-Anforderung in die Warteschlange und sendet nichts. A befindet sich im Zustand "Angefordert". A vergleich die Zeitstempel B(21) größer A(12). A gewinnt. A stellt B-Anforderung in seine Warteschlange.
(d) C verläßt kritischen Bereich und sendet eine Bestätigung für jede Anforderung in seiner Warteschlange. A erhält eine Bestätigung von C und tritt in den kritischen Bereich ein.
(e) A verläßt kritischen Bereich, in seiner Warteschlange ist nur eine Anforderung von B, deswegen sendet A eine Besätigung an B. B hat nun alle Bestätigungen gesammelt und kann in den kritischen Bereich eintreten.

stellt der Prozess die eingehende Anforderung in die Warteschlange und sendet nichts. Sobald alle Berechtigungen eingetroffen sind, kann der Prozess in den kritischen Bereich eintreten.

– Zustand "Kritisch": Wenn neue Anforderungsnachricht eintrifft, stellt der Prozess die eingehende Anforderung in die Warteschlange und sendet nichts. Wenn der Prozess sich entscheidet den kritischen Bereich zu verlassen, entfernt er Anforderungen aus der Warteschlange und sendet eine OK-Nachricht an die entsprechenden Prozesse.
  Abb. 22 zeigt ein mögliches Szenario für ein verteiltes System aus 3 Prozessen. Dieser Algorithmus soll in folgenden Kapiteln mit Sequenzdiagramm, Petrinetz und Business Process Diagram modelliert werden.

4.1 Sequenzdiagramm

Ein Sequenzdiagramm modelliert immer nur einen möglichen Pfad im Ablauf. Die Anzahl aller möglicher Pfade im System kann aber sehr hoch werden. Sequenddiagramme sind deswegen ein einfaches grafisches Mittel, die grobe Idee des Algorithmus oder eines komplizierten Ablaufes zu illustrieren. Der Schwerpunkt liegt auf der Interaktion der Kommunikationspartner. Abb. 23 zeigt drei Prozesse im verteilten System und wie diese untereinander Nachrichten austauschen. Der Inhalt der Nachrichten, also die Daten, kann man auf den Nachrichtenlinien angeben. Unter Ereignissen versteht man bei Sequenzdiagrammen das Senden oder das Empfangen von Nachrichten. Die nebenläufige Ereignisse lassen sich einfach Ablesen, siehe dazu Abb. 23. Sequenzdiagramme haben kein formales mathematisches Modell als Grundlage, deswegen lassen sich keine Korrekteitsbeweise über den Algorithmus machen. Die Manipulatin von Daten ist kaum modellierbar. Die Operationen, die die Prozesse durchführen, lassen sich nur als Textkommentare angeben. Der größte Nutzen von Sequenzdiagrammen liegt in der Lesbarkeit und Verständlichkeit. Die Diagramme sind auch für Laien verständlich, z.B. für Endbenutzer, Geschäftspartner etc.

4.2 Petrinetz

Für die Praxis sind Petrinetze mit "schwarzen Marken" in vielen Fällen nicht ausreichend. In sogenannten *High-level-Petrinetzen* können Marken selbst Informationen tragen. Die Marken sind vom bestimmten Typ, wie in der Programmierung(z.B. bool, int, zusammengesetzte Datentypen(records) etc.). Die Kantenbeschriftung legt fest, welche Marken beim Schalten einer Transition konsumiert und produziert werden. Als Beschriftung sind Variablen erlaubt. Transitionen erhalten boolische Ausdrücke. Eine Transition kann nur dann schalten, wenn der Ausdruck zum boolischen Wert "True" evaluiert. In Abb. 24 ist ein High-Level-Netz zu sehen, der verteilten wechselseitigen Ausschluss realisiert. Linkes Teilnetz modelliert Prozess A, rechtes Teilnetz Prozess B. Das Netz zeigt die Situation im verteilten System wo A und B gleichzeitig in den kritischen Bereich eintreten wollen. Es gibt zwei Typen von Marken: Marken, die eine Bedingung und Marken, die eine Nachricht repräsentieren. Die Stellen sA4, sA5, sA2, sA7,

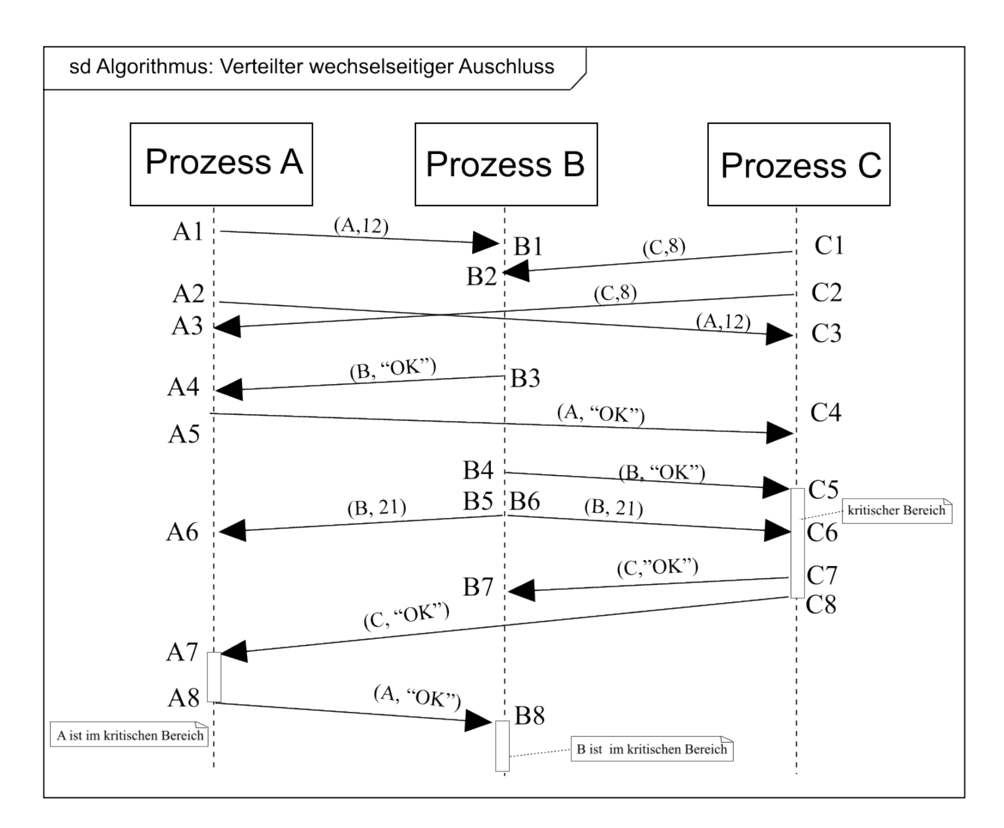


Abbildung 23. Sequenzdiagramm. Ein mögliches Szenario für den Ablauf des verteilten wechselseitigen Ausschlusses. Nebenläufige und **kausale Beziehungen** zwischen Ereignissen sind leicht zu erkennen: es sind genau die Ereignisse z.B. zu B3 nebenläufig, von denen (bzw. zu denen) keine Information entlang der Pfeile nach (bzw. von) B3 fließt. Nebenläufige Ereignisse zu B3: {A2, C2, C3}. B3 ist dagegen von A1 kausal abhängig, A1 ist die Ursache für B3.

sB3, sB5, sB4 nehmen Marken vom Typ "Nachricht" auf. Die anderen Stellen sA1, sA6, sA8, sA10 und die spiegelgleichen Stellen vom Prozess B, nehmen Marken vom Typ "bool" auf. Die Kante zwischen sA9 und tA6 ist die sogenannte **Inhibitor-Kante**. Diese Art von Kanten ist eine Erweiterung von klassischen Petrinetzkanten. Diese Kante ermöglicht es eine Stelle auf Leerheit zu prüfen, was oft nützlich ist. Transition tA6 ist nur dann aktiviert, wenn die Stelle sA9 **leer** ist und die Stelle sA10 eine Marke hat. tA6 schaltet also nur wenn der Prozess den kritischen Bereich verlassen hat und die Warteschlange leer ist.

Petrinetze werden zur Modellierung und Simulation von allgemeinen Vorgängen insbesondere von Arbeitsabläufen verwendet [9, Seite 490]. Petrinetze besitzen jedoch eine **statische Natur**. Will man z.B. das Netz in Abb. 24 auf mehr als 2 Prozesse erweitern, muß man mehr als 20 neue Knoten einfügen und außerdem das vorhandene Netz entsprechend anpassen. Die Lesbarkeit und Verständlichkeit sinkt mir der wachsenden Zahl der Prozesse. Ein weiterer großer Nachteil von Petrinetzen, dass sie keinen Konzept der Zeit kennen. Ausführliche Informationen zu Petrinetzen finden Sie in [10] und [1].

4.3 Business Process Modeling Notation

Abb. 25 und Abb. 26 sind zwei Variationen, die ein und denselben Algorithmus modellieren: verteilter wechselseitiger Ausschluss. Variante Nr.1 verwendet Intermediate Events mit dem Trigger "Nachricht" (Briefcouvert im Kreis), um die eintreffenden Nachrichten zu modellieren. Variante Nr.2 in Abb. 26 verwendet Pools, um die Interaktionen zwischen Prozessen deutlich zu machen. Beide Varianten sind unendliche Prozesse, die vom Start Event irgendwann gestartet werden und danach unendlich lange laufen. Wann und von wem der Start Event ausgelöst (getriggert) wird ist aus dem Diagramm nicht ersichtlich. Nachdem das Start Event ausgelöst wird, startet die Task "Zustand Ruhe", der Prozess geht in den Zustand "Ruhe" über. Task Task "Zustand Ruhe" wartet auf neue Nachrichten von anderen Prozessen. Diese Task kann auf zwei Wegen beendet werden: (1) wenn neue Nachricht eintrifft oder (2) wenn sich der Prozess irgendwann entscheidet in den kritischen Bereich einzutreten. Angenommen (2) passiert: Task "sende Nachricht..." staret und sendet an jeden Prozess im System eine Anforderungsnachricht. Wenn Task "sende Nachricht..." beendet wurde, wandert der Kontrollfluß zum Intermediate Event mit dem Trigger "Message", hier wartet der Prozess auf Nachrichten. Trifft neue Nachricht ein, geht der Fluß zum Exclusive Decision gateway an dem 3 Bedingungen an ausgehenden Pfaden stehen. Der Pfad dessen Bedingung zu "True" ausgewertet wird, wird als Richtung für Kontrollfluss gewählt.

Das Intermediate Event mit dem Trigger "Link" (scharzer ausgefüllter Pfeil) ist ein "Go To"-Konstrukt, mit dem von einem zum anderen Link-Event (schwarzer weißer Pfeil) des Diagramms gesprungen werden kann.

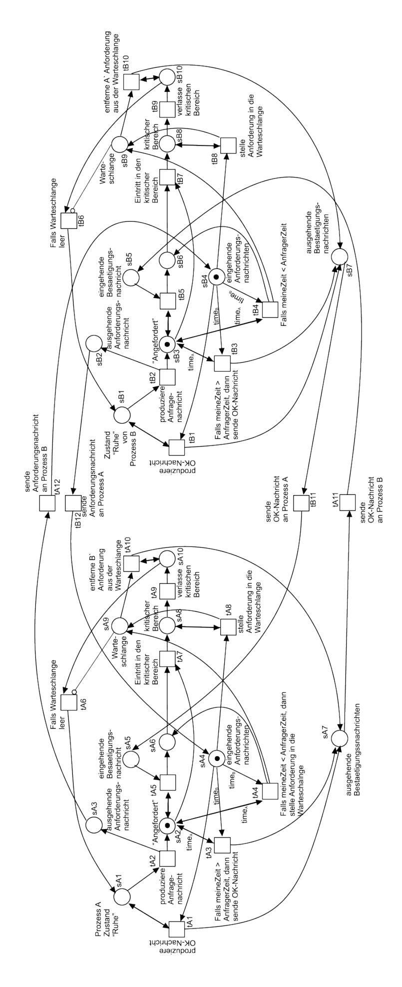


Abbildung 24. High-Level-Petrinetz mit der Markierung {sA2,sA4,sB3,sB4}. Linkes Teilnetz modelliert Prozess A, rechtes Teilnetz Prozess B. Das Netz zeigt die Situation im verteilten System wo A und B gleichzeitig in den kritischen Bereich eintreten wollen.

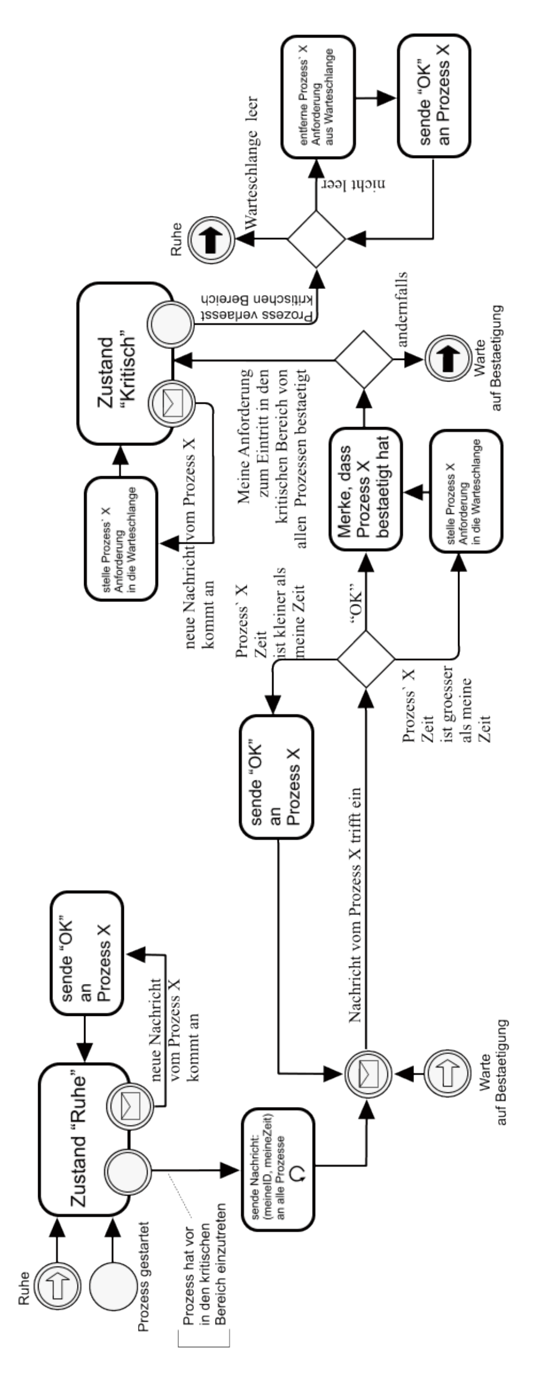


Abbildung 25. BPMN. Verteilter wechselseitiger Ausschluss. Variante 1

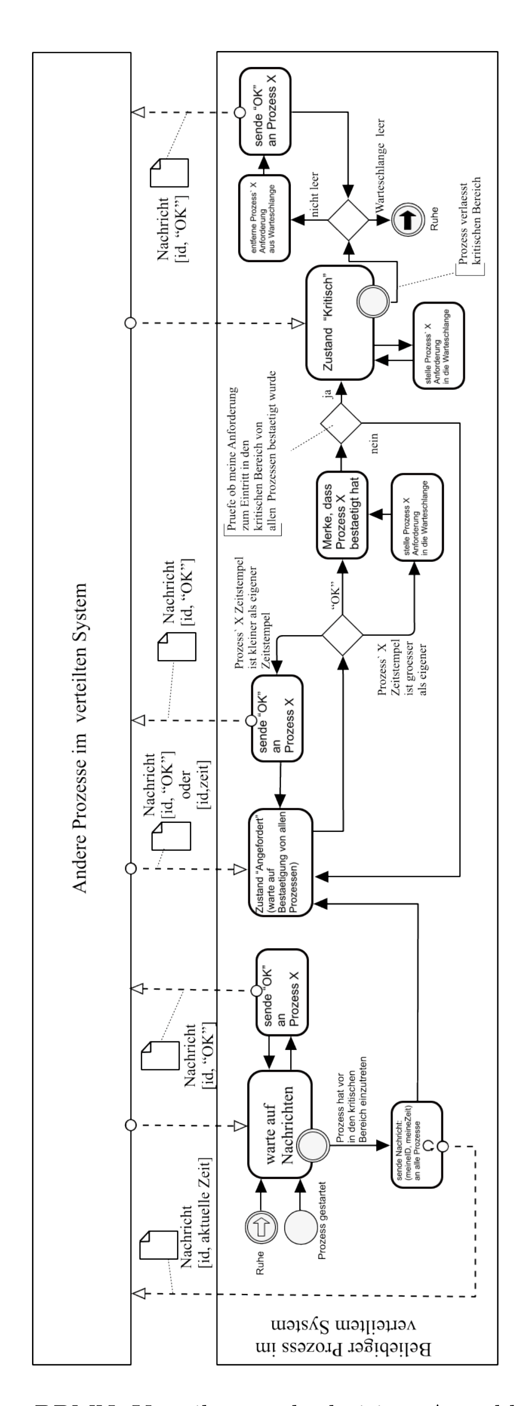


Abbildung 26. BPMN. Verteilter wechselseitiger Ausschluss. Variante 2

5 Schluss

Jedes Modell ist eine Vereinfachung der Wirklichkeit. Viele Datails läßt man weg, dadurch wird die Komplexität des Problems reduziert und gleichzeitig wächst die Übersichtlichkeit und die Verstädnlichkeit des Modells. Welche Details wesentlich und welche unwesentlich sind, hängt von der gewünschten Perspektive, vom Blickwinkel auf das Problem. Bei den Sequenzdiagrammen werden die Interaktionen zwischen den Kommunikationspartnern als wesentlich betrachtet. Nach welchen Regeln die Nachrichten gesendet werden, läßt sich aud den Sequenzdiagrammen nicht ablesen. Bei Business Process Diagrammen (BPD) werden wie der Name schon sagt, die Organisationsabläufe als wesentlich betrachtet. Die Kommunikation zwischen den Prozessen in verteilten Systemen läßt sich mit Pools darstellen. Die Übersichtlichkeit der Diagramme sinkt allerdings drastisch, wenn mehr als 2 kommunizierende Prozesse modelliert werden sollen.

Viele Systeme sind reaktiv, d.h. sie müssen auf "Umweltreize" reagieren, um irgendwelche Aktionen anzustoßen (triggern). Auslöser für Events können z.B sein: ankommende Nachrichten, Timeouts, Exceptions etc. Sehr praktisch erweisen sich dabei Events bei BPD. Es sind nur wenige Trigger standardisiert, die Entwickler dürfen allerdings eigene Trigger einführen.

Alle in dieser Arbeit vorgestellten Diagramme können sehr beschränkt die Transformation von Daten modellieren. In Petrinetzen lassen sich Daten mit einigen Erweiterungen realisieren. Die klassischen Petrinetze wurden im Laufe der Jahre um viele Konzepte erweitert: Inhibitor-Kanten, Farbige Petrinetze, High-Level-Petrinetze etc. Bei Business Process Diagrammen kann man die Daten mit Data Objects darstellen, die aber eine zweitrangige Rolle spielen und meistens optional sind.

Petrinetze besitzen eine solide mathematische Basis, deswegen können Algorithmen und Abläufe auf Korrektheit und Fehlerfreihet getestet werden (z.B. mit Hilfe von Erreichbarkeitsgraphen, S- oder/und T-Invarianten).

Business Process Modeling Notation hat eine Reihe von Vorteilen gegenüber UML wenn es um Modellierung von Geschäftsprozessen geht: kompaktere Flußmodellierungstechnik, solide mathemiatische Basis (Pi-Calculus). Business Process Diagramme können außerdem auf Execution Languages (z.B. BPEL4WS) abgebildet und simuliert werden.

Literatur

[1] Baumgarten. *Petri-Netze. Grundlagen und Anwendungen.* BI-Wissenschaftsverlag, 1990.

[2] M. H.-P. Gumm. *Einführung in die Informatik.* Odenbourg Verlag, 2002.

[3] http://de.wikipedia.org/wiki/Bildfahrplan. Bildfahrplan, Februar 2008.

[4] Object Management Group. Business Process Modeling Notaion specification, February 2006.

[5] Object Management Group. Omg unified modeling language, superstructure, v2.1.2, November 2007.

[6] G. P. Peter Rechenberg. *Informatik-Handbuch.* Hanser, 2002.

[7] A. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.
[8] A. S. Tanenbaum and M. von Steen. *Verteilte Systeme. Grundlagen und Paradigmen.* Pearson Studium, 2003.
[9] P. C. und Prof.Dr.Andreas Schwill. *Duden Informatik. Ein Fachlexikon für Studium und Praxis.* Dudenverlag, 2003.
[10] P. W. Vogler. Petrinetze. (vorlesungsskript), SS2007.

Eigenschaften dezentraler Koordinationsmechanismen

Michael Boegler

Universität Augsburg
michael.boegler@student.uni-augsburg.de

Zusammenfassung Dezentrale Koordination spielt in heutigen Systemen, die stetig größer, dynamischer und unübersichtlicher werden, eine immer wichtigere Rolle. Ob in Sensornetzen, mobilen Ad-Hoc Netzen, im Netzwerk-Topologie Management, oder im Peer-to-Peer Bereich, eine zentrale Kontrolle ist praktisch nicht, oder nur mit unverhältnismäßig großem Aufwand, realisierbar. In dieser Arbeit werden dezentrale Mechanismen aufgezeigt, die die Koordination der einzelnen Teilnehmer regeln. Zur Koordination können die Agenten gänzlich unterschiedliche Algorithmen verwenden, deren Grundgedanken oft aus der Natur stammen. Neben den, von der Natur inspirierten Verfahren, werden in dieser Arbeit auch Koordinationsmöglichkeiten beschrieben, die ihren Ursprung in der Volkswirtschaftslehre haben. Diese Mechanismen haben zum Ziel, dass eine Vielzahl von unabhängig agierenden Agenten zusammenarbeitet, um ein angestrebtes, systemweites Verhalten zu erreichen. Hierfür sammeln die Agenten lokale Informationen, verarbeiten diese und reagieren darauf. Das Gesamtsystem kann dadurch ein Verhalten erlangen, das den einzelnen Agenten nicht einprogrammiert wurde und zu einer Selbst-Optimierung, Selbst-Konfiguration oder ähnlichen selbst-x Eigenschaften des kompletten Systems führen kann.

1 Einleitung

Diese Arbeit beginnt in Kapitel 2 mit einer Begriffsklärung für dezentrale Koordination. In Kapitel 3 und 4 werden einige ausgewählte Verfahren der dezentralen Koordination erläutert. Grundsätzlich werden die Koordinationsmöglichkeiten in dieser Arbeit in zwei Bereiche eingeteilt, die jedoch in der Praxis fließend ineinander übergehen. In Kapitel 3 wird beschrieben, wie die Teilnehmer eines Systems durch die Umgebung koordiniert werden. Im Kapitel 4 wird beschrieben, wie die Agenten sich untereinander koordinieren, was wiederum direkt oder auch indirekt über die Umgebung geschehen kann. Bei diesem Verfahren übernehmen die Agenten selbst die Koordination. Bei beiden Koordinationstypen müssen die Umgebung, sowie auch die Agenten auf das jeweilige Modell abgestimmt sein. Diese verschiedenen Koordinationsalgorithmen sind für unterschiedliche Anforderungsmodelle geeignet. Abschließend erfolgt eine Übersicht über alle vorgestellten dezentralen Koordinationsmechanismen mit den jeweils realisierten selbst-x Eigenschaften.

2 Begriffsklärung

Einige Begriffe, die in dieser Arbeit verwendet werden, sind in der Literatur nicht eindeutig definiert. Um den daraus möglicherweise resultierenden Missverständnissen entgegen zu wirken, wird in diesem Abschnitt eine Definition gewählt, wie die einzelnen Begriffe in dieser Arbeit zu verstehen sind.

Weitere Begriffe werden der Einfachheit halber hier erklärt, da an mehreren Stellen der Arbeit darauf verwiesen wird, bzw. die Begriffe zum Verständnis der einzelnen Verfahren benötigt werden.

2.1 Dezentrale Kontrolle - Zentrale Kontrolle

Laut [2] bedeutet dezentrale Kontrolle, nur lokale Mechanismen zu verwenden, um globales Verhalten zu beeinflussen. Es gibt keine zentrale Kontrolle. Diese Art von Kontrolle ist vor allem für immer komplexer und dynamischer werdende Systeme nötig, da eine von einem zentralen Koordinator ausgehende Kontrolle einen enormen Kommunikationssaufwand erfordert und somit Zeit und Geld kostet. Es ist durch die Dynamik der Systeme oft nicht möglich einen zentralen Koordinator einzusetzen. Des Weiteren müssen moderne Systeme teilweise selbstständig handeln, um auf unvorhersehbares Verhalten reagieren zu können.

Zentrale Kontrolle hingegen bedeutet, dass ein zentraler Koordinator die Kontrolle übernimmt. Er trifft Entscheidungen für alle Teilnehmer des gesamten Systems. Das macht ihn in diesem zentralen Fall zu einem single point of failure. Wenn er ausfällt, geht die Kontrolle des gesamten Systems verloren. Dezentrale Verfahren haben dieses Problem nicht, was das System fehlerunanfälliger macht.

2.2 Agenten

Agenten sind die Aktuatoren in einem dezentralen System. Sie übernehmen die Koordination und/oder werden koordiniert. Wie in [1, 5, 2] beschrieben, sammelt jeder Agent Informationen aus seiner lokalen Umwelt und trifft daraufhin, unabhängig von anderen Agenten, Entscheidungen. Diese Entscheidungen hängen von lokalen Entscheidungsparadigmen ab. Die Kommunikation jedes Agenten mit seiner Umwelt kann sich sowohl auf die Umgebung, als auch auf andere Agenten beziehen, von denen er z.B. lernt oder mit denen er handelt. Agenten können z.B. Netzwerkkomponenten, oder auch dynamische Objekte zur Lösung eines Problems sein.

2.3 Emergenz

Diese Arbeit handelt von dezentraler Koordination, d.h. dem Fehlen eines zentralen Koordinators, wodurch das System als Ganzes nicht kontrolliert werden kann. Wie bei den Agenten (siehe Abschnitt 2.2) beschrieben, kann nur das Verhalten der einzelnen Agenten beeinflusst werden. Ein aus deren Interaktionen entstehendes globales Verhalten wird in dieser Arbeit als emergent bezeichnet. Laut [9] ist Emergenz folgendermaßen zu verstehen:

Emergenz [lateinisch] die,
Wissenschaftstheorie: Das Auftreten neuer, nicht voraussagbarer Qualitäten beim Zusammenwirken mehrerer Faktoren.

Auch aus dieser Definition kann man erkennen, dass ein Verhalten gemeint ist, das aus den Aktionen der einzelnen Agenten als Gesamtes entsteht. Dies kann auch als eine Art Synergie-Effekt bezeichnet werden.

2.4 Selbst-Organisation

Die Emergente Eigenschaft der Selbst-Organisation eines Gesamtsystem besteht laut [10] aus vier Basiseigenschaften: Selbst-Konfiguration, Selbst-Optimierung, Selbst-Heilung und Selbst-Schutz.

Unter Selbst-Konfiguration versteht man laut [10], dass sich das System gemäß eines Ziels selbst konfiguriert. Es wird angegeben welche Konfiguration angestrebt wird, jedoch nicht wie die Durchführung zum Erreichen des Ziels aussehen soll. Ein Beispiel für Selbst-Konfiguration ist eine Software die sich, unabhängig vom Betriebssystem, selbst installiert. Als Ergebnis soll die Software installiert sein. Die Duchrführung, welche Pakete installiert werden müssen, abhänging von der Umgebung, entscheidet die Software jedoch selbst.

Selbst-Optimierung bedeutet laut [10], dass ein autonomes Computersystem den Ressourcenbedarf optimiert. Das System kann aus Eigeninitiative entscheiden, dass eine Änderung vorgenommen wird, um die Leistungsfähigkeit zu verbessern.

Wenn ein autonomes Computersystem Probleme erkennen, diagnostizieren und beheben kann, spricht man laut [10] von Selbst-Heilung. Probleme können hierbei kleinere Fehler wie z.B. ein Bit Fehler in einem Speicher Chip (Hardwarefehler) oder schwerwiegendere Probleme wie z.B. fehlerhafte Einträge im Verzeichnisdienst (Softwarefehler) sein. Das System versucht den Fehler zu beheben, wobei die Methode zur Fehlerbehebung dem System selbst überlassen bleibt. Es kann z.B. auf eine redundante Komoponente wechseln, oder ein Softwareupdate installieren. Wichtig ist, dass als Ergebnis des Heilungsprozesses das System nicht mehr beschädigt ist, nur dann spricht man von Selbst-Heilung. Diese Fehlertolleranz ist ein wesentlicher Aspekt der Selbst-Heilung.

Selbst-Schutz bedeutet laut [10], wenn sich ein System vor bösartigen Angriffen oder Benutzern, die aus Versehen Änderungen in der Software vornehmen, z.B. wichige Dateien löschen, schützt. Das System verändert sich sofort, um Sicherheit, Datenschutz und Datensicherheit zu gewährleisten. Sicherheit ist ein wichtiger Aspekt des Selbst-Schutzes.

2.5 Selbst-anpassend - anpassungsfähig

Laut [5] liegt der Unterschied der beiden Begriffe darin, dass anpassungsfähige Systeme an ihre eingesetzte Umgebung angepasst werden können, wohingegen sich selbst-anpassende Systeme, ohne Benutzereingriff, an ihre Umgebung anpassen. Angepasst werden anpassungsfähige Systeme durch einen externen Benutzer, indem dieser die zur Verfügung gestellten Interfaces nutzt. Selbst-anpassende

Systeme jedoch passen sich selbstständig an. Diese Anpassungen basieren auf wahrgenommenen Veränderungen des System- oder Umgebungszustandes.

2.6 Feedback

Ein Verfahren zur Regulierung in dezentralen Systemen, ist das Feedback Prinzip. Es gibt zwei verschiedene Arten, positives und negatives Feedback. Positives Feedback wirkt als Verstärkung, negatives Feedback als Bremse oder Regulator. Als Beispiel wird in [5] ein Peer-to-Peer Netz aufgeführt. Je mehr Teilnehmer ein Peer-to-Peer Netz hat, desto attraktiver wird es für neue Peers, was zu einer Erhöhung der Teilnehmer führt. Diesem positiven Feedback wirkt das negative Feedback entgegen, das durch die schlechtere Performance des immer größer werdenen Netzes entsteht. Dieses Zusammenspiel führt zur Selbst-Regulierung des gesamten Systems.

3 Umgebungsgesteuerte, dezentrale Koordination mit Gradientenfeldern

Bei der umgebungsgesteuerten dezentralen Koordination wird die Führung der Agenten von der Umgebung vorgenommen. Auch die Koordinationsstruktur wird von der Umgebung und nicht von den Agenten selbst erstellt. Die Teilnehmer folgen lediglich einer Art rotem Teppich [3] der von der Umgebung erstellt wurde. Eine Möglichkeit für die Koordination dieser Art sind Gradientenfelder, auch Verlaufsfelder genannt.

Ziel der Gradientenfelder ist es laut [4], mehrere autonome Agenten in einer Umgebung zu einer räumlich zusammenhängenden Bewegung zu bringen um, sie zu koordinieren. Das Vorbild der Gradientenfelder Methode stammt aus der Physik. Wie in Abbildung 1 zu sehen ist, gibt es eine Quelle, von der aus sich kreisförmig das Gradientenfeld ausbreitet. Diese Quelle kann z.B. mit einem Magneten verglichen werden, wobei die Kreise dem Magnetfeld um die Quelle entsprechen. Die Agenten werden entweder vom Magneten angezogen und bewegen sich zur Quelle hin oder werden abgestoßen und bewegen sich von der Quelle weg.

Da das Verlaufsfeld von der Umgebung initiiert wird, handelt es sich hier um eine umgebungsgesteuerte, dezentrale Koordination. Die Feldstärken können je nach Implementierung, von der Quelle aus gesehen, zu- oder abnehmen. Die implementierte Umgebung ist hierbei für die Ausbreitung der Wellen verantwortlich. Die einzelnen Verlaufsfelder enthalten die jeweilige Stärke des Feldes im aktuellen Gradienten. Der nächste Gradient hat dann, je nachdem, ob ansteigende oder abfallende Werte gewünscht sind, eine höhere oder niedrigere Intensität. Die Erhöhung oder Verringerung erfolgt an den einzelnen Gradienten-Objekten die, weiter von der Quelle entfernt, auf dem nächsten Gradienten liegen. Dadurch wird ein konstantes, abfallendes, oder ansteigendes Verlaufsfeld erzeugt.

Die Agenten überwachen ihre Umgebung, erkennen dadurch wo sich die Quelle befindet und entscheiden daraufhin selbstständig, wohin sie sich als nächstes

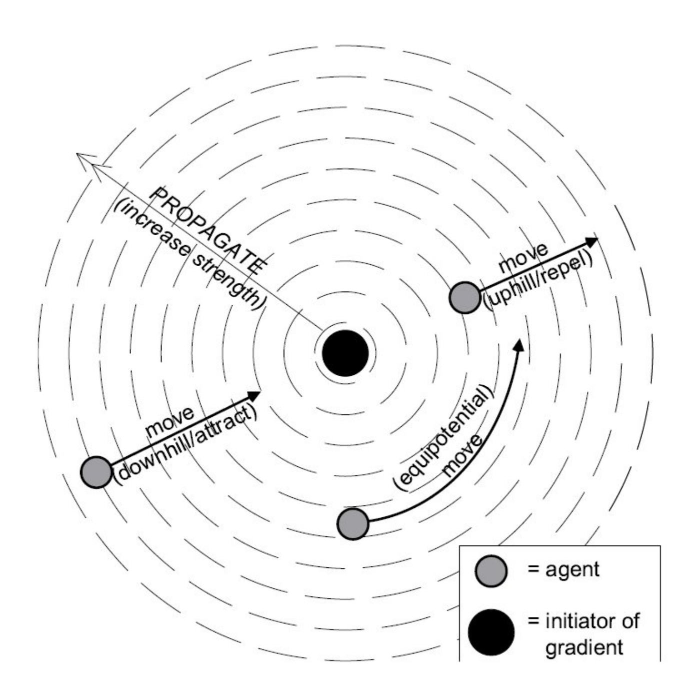


Abbildung 1. Beispiel eines Gradientenfeldes aus [4]

bewegen. Sie können sich nicht nur auf die Quelle zu oder von ihr weg bewegen, sondern haben auch die Möglichkeit, wie in Abbildung 1 zu sehen, sich äquipotential, auf der gleichen Verlaufswelle bleibend, zu bewegen. Durch die Einführung mehrerer Quellen, werden mehrere Verlaufsfelder initiiert, wodurch die Agenten sich entscheiden können, auf welche Quelle sie reagieren möchten.

Als praktisches Beispiel wird der Geruch eines Brathähnchen- und Dönerstandes genommen. Die Agenten sind in diesem Fall Menschen, die entweder einen Döner, oder ein Brathähnchen essen möchten. Der Geruch, der von beiden Ständen ausgeht, wird über die Umgebung, in diesem Fall die Luft, übertragen. Es ist ersichtlich, dass die Intensität der Verlaufswellen, in diesem Beispiel der Geruch, über die Distanz abnimmt. Ebenfalls kann ein Mensch die beiden Gerüche unterscheiden. Wenn ein hungriger Kunde ein Brathähnchen möchte, wird er sich in Richtung Brathähnchenstand bewegen. Er wird versuchen, den kürzesten Weg zu wählen und folgt dem immer stärker werdenden Geruch der Brathähnchen, bis er sein Ziel erreicht. Nach dem gleichen Prinzip wird ein Mensch vorgehen, der einen Döner bevorzugt. Beide werden den kürzesten Weg zu ihrem Ziel nehmen, und sich nicht von dem jeweils anderen Geruch irritieren lassen. Die direkte Bewegung zum Stand hin, kann als gieriges Verhalten bezeichnet werden. Beide Menschen haben einen lokalen Plan, etwas zu essen, der sich nur in der Präferenz der Nahrung unterscheidet. Es werden auch nur Informationen aus der direkten Umgebung verwendet, nämlich der Geruch in der

Luft. Das Ziel wird jeweils erreicht, indem Sie dem roten Teppich, dem immer stärker werdenden Duft, folgen. Man kann sich in diesem Zusammenhang auch vorstellen, dass ein Mensch der Dönergeruch nicht leiden kann, sich so weit vom Dönerstand entfernen wird, bis er den Geruch nicht mehr wahrnimmt. Es kommt auf die Präferenzen des Agenten an, ob er sich in Richtung der Quelle, oder von ihr weg bewegt. Grundsätzlich jedoch haben alle Agenten das gleiche Verhalten.

Eigenschaften der Gradientenfelder:

- Durch eine Bewegung der Quelle erfolgt eine globale Selbst-Organisation. Je nachdem wie ein Agent auf die Gradienten der Quelle reagiert, folgt er der sich bewegenden Quelle oder bewegt sich von ihr weg.
- Da die Teilnehmer gierig sind, finden sie selbstständig den kürzesten Weg zum Initiator der Gradienten.
- Ziel des Verlaufsfeldes sollte sein, dass die Agenten an der Quelle die Lösung für das aktuelle Problem finden.
- Erhöhungen der Agentenanzahl können vom System ohne Probleme verarbeitet werden.
- Veränderungen der Gradienten, z.B. das Stoppen des Aussendens der Gradienten, können von diesem Koordinationsmechanismus verarbeitet werden.

Da es sich um eine umgebungsgesteuerte, dezentrale Koordination handelt, ist die Logik in den Agenten bei dieser Methode sehr einfach gehalten, wohingegen die Umgebung relativ komplex ist. Die Gradienten müssen sich über das System erstrecken, und zu- bzw. abnehmen können. Die Umgebung erstellt somit den roten Teppich, dem die Agenten nur folgen müssen.

Mögliche Einsatzgebiete finden sich laut [3] im Netzwerkrouting wie auch im städtischen Stau-Management.

4 Agentengesteuerte, dezentrale Koordination

Im Gegensatz zur umgebungsgesteuerten, dezentralen Koordination (siehe Kapitel 3) übernehmen im agentengesteuerten, dezentralen Fall die Teilnehmer selbst die Koordination. Je nach Modell kommunizieren die Agenten direkt (Kapitel 4.3 Marktbasierte Kontrolle) oder über die Umgebung (z.B. Digitaler Pheromonpfad aus Abschnitt 4.2) miteinander. Je nach Algorithmus muss die Umgebung bzw. der Agent fähig sein, diese Kommunikationsart zu unterstützen.

Aufgrund der Komplexität werden nur einige ausgewählte Verfahren beschrieben, die einen Überblick über die Vielzahl der Möglichkeiten geben sollen. In [3] werden detailliert weitere Möglichkeiten vorgestellt, wie für verschiedene Anwendungsgebiete dezentrale Koordination ermöglicht werden kann.

4.1 Gemeinschaftlich verstärktes Lernen

In der Literatur wie auch im weiteren Verlauf der Arbeit wird gemeinschaftlich verstärktes Lernen mit CRL abgekürzt. Dies leitet sich vom englischen, collabo-

rative reinforcement learning, ab. Laut [6] ist CRL ein dezentraler Ansatz zur Erstellung und Aufrechterhaltung von systemweiten Eigenschaften in dezentralen Systemen.

CRL basiert auf RL, dem reinforcement learning. RL bezieht sich auf einzelne Agenten, die wie in [6] und [5] beschrieben, durch die Versuch-und-Irrtum Methode mit der Umgebung kommunizieren. Bei der Versuch-und-Irrtum Methode werden laut [7]

> „[...] zufällig ausgewählte Techniken zur Lösung einer Aufgabe ausprobiert, sich erfolgreiche Handlungsweisen gemerkt und bei späteren Gelegenheiten erneut angewendet. “

Das Ziel eines RL Agenten ist es, durch die Auswahl der richtigen Aktionen die Verstärkung seiner lokalen Präferenzen über einen Zeitraum zu maximieren.

Da die Auswahl der Aktionen von Wahrscheinlichkeiten abhängt, entsteht die schon genannte Versuch-und-Irrtum Methode. Dies ist für verteilte Systeme kein angemessenes Verfahren um selbst-x Verhalten zu erlernen, da vor allem Systeme, die in Echtzeit laufen, eine suboptimale Aktionsauswahl nicht dulden.

Daher erweitert CRL laut [6] RL um ein Koordinationsmodell, das auf einer Variante des Schwarm-Intelligenz-Algorithmus (siehe [8]) basiert, bei dem Agenten mit ihren direkten Nachbarn kommunizieren und vom Erfolg ihrer Nachbarn lernen. Wie bei einem dezentralen Ansatz üblich, gibt es kein globales Wissen. Jeder Agent weiß nur über sich und seine Kommunikationsnachbarn bescheid. Um systemweite Optimierungsprobleme zu lösen, beginnt jeder Agent sein Diskretes Optimierungsproblem (DOP) mit Hilfe von RL zu lösen. Das Ergebnis, z.B. die Kosten für Problemlösungen, wird den Nachbarn zur Verfügung gestellt. Deren Ergebnisse werden ebenfalls gespeichert, wobei diese Informationen über eine bestimmte Zeit abklingen. Damit wird modelliert, dass Informationen nur eine bestimmte Zeit gültig sind. Somit hat jeder Agent einen Überblick, welcher Nachbar, welches Problem, mit welcher Effizenz lösen kann und wie alt dieser Lösungsansatz schon ist. Die DOP können so an andere Agenten weitergegeben werden, damit diese das Problem effizienter lösen.

Durch die Kommunikation mit den Nachbarn können die Agenten erkennen, wer von ihren Nachbarn eine bessere Problemlösung gefunden hat. Ein Agent kann versuchen die optimierte Methode seines Nachbarn zu verwenden um sein DOP zu lösen. Dies wird als gemeinschaftliches Feedback bezeichnet. Sollte das DOP durch die neue Methode besser gelöst werden, kann ein positives Feedback erzeugt werden (siehe Abschnitt 2.6). Dadurch erkennt die Gruppe der Nachbaragenten, dass mit diesem verbesserten Lösungsansatz das Problem effizienter gelöst werden kann. Es wird das eigene Verfahren verbessert, indem von den Methoden der Nachbarn gelernt wird. Über die Zeit hinweg wird immer die optimalste Problemlösung aus einer Reihe von Lösungsversuchen der Nachbaragenten, für ein lokales Problem ausgewählt. Das erzeugte positive Feedback kann laut [6] zu einer emergenten Eigenschaft, der Selbst-Optimierung (siehe Abschnitt 2.3) des Systems führen. Jeder Agent selbst und somit das System als Ganzes, trifft für unterschiedliche DOP optimale Entscheidungen. Negatives

Feedback können entweder Beschränkungen im System sein, oder das abklingen der gespeicherten Lösungsansätzte der Nachbarn. Durch das positive und negative Feedback nähern sich die Agenten des Systems so einem stabilen Zustand.

Veranschaulicht kann man wie in [6] beschrieben annehmen, dass die Agenten einzelne Netzwerkkomponenten sind, die Aufgaben erfüllen und ihre Kosten dafür veröffentlichen. Im Netz werden Aufgaben an die Komponenten verteilt, die diese am kostengünstigsten lösen können. Sollte eine Verbindung zu einer Komponente abbrechen, werden die Informationen über die Bearbeitungskosten, welche die Nachbarn gespeichert haben, langsam abklingen. Die Nachbarn „vergessen“, dass sie an diese Komponente ihre Aufgaben schicken können, da sie sich nicht mehr meldet. Dieses nicht einprogrammierte Verhalten ist eine Konsequenz des negativen Feedbacks und führt zu einer Selbst-Organisation des Systems.

Da es sich um eine Agentengesteuerte dezentrale Koordination handelt ist die Komplexität der Agenten, größer als die der Umgebung. Sie müssen einerseits ihre DOP lösen können, ihre Lösungen veröffentlichen und entscheiden ob sie ihr Problem zur Lösung an einen Nachbarn weitergeben. Die Umgebung in der sich die Agenten bewegen, bedarf keiner besondern Logik.

Einsatzgebiete sind laut [6] z.B. das Load Balancing in einem Netzwerk.

4.2 Digitaler Pheromonpfad

Laut [3] wird der Algorithmus verwendet, voneinander unabhängige Agenten dezentral zu koordinieren, um ein gemeinsames Ziel zu erreichen. Es werden nur lokale Informationen verwendet, um dieses globale Ziel zu erreichen.

Das Vorbild für diesen Algorithmus kommt aus der Natur. Es wird beispielsweise das Verhalten von Ameisenkolonien auf ihrer Suche nach Nahrung nachgebildet. Im Gegensatz zu den Gradientenfeldern aus Kapitel 3, übernehmen hier die Agenten die Koordination um zum Ziel zu gelangen. Die Kommunikation erfolgt jedoch nicht direkt wie bei CRL (siehe Abschnitt 4.1), sondern die Umgebung muss eine indirekte Kommunikation ermöglichen. Diese Kommunikationsform wird in der englischsprachigen Literatur als Stigmergy bezeichnet. Stigmergy bedeutet, dass die Agenten miteinander kommunizieren, indem sie ihre momentane lokale Umgebung beeinflussen, bzw. verändern [3].

Bei den Ameisen erfolgt diese indirekte Kommunikation mittels Pheromonen. Sobald eine Ameise Nahrung gefunden hat, begibt sie sich damit zurück zum Nest und hinterlässt auf ihrem Weg in bestimmten Abständen Duftstoffe, die anzeigen, dass eine Nahrungsquelle gefunden wurde. Diese Pheromone verflüchtigen sich mit der Zeit. Andere Ameisen können die Stärke und den Verlauf der Pheromone erkennen und daraufhin ihr bisheriges Verhalten ändern, und dem Pfad in Richtung Nahrung folgen. Auch sie werden auf dem Rückweg zum Nest Pheromone freigeben, was dazu führt, dass der Pheromonduft auf dem Pfad verstärkt wird. Je mehr Ameisen nun diesem Pfad folgen, desto intensiver wird auch der Duft des Pfades. Da sich die Pheromone mit der Zeit verflüchtigen, führt das, sobald keine Nahrung mehr vorhanden ist und somit keine Ameisen mehr Pheromone auf dem Pfad hinterlassen, zur Abnahme des Duftpfades, bis

dieser komplett verschwunden ist. Dadurch wird verhindert, dass weitere Ameisen diesem Pfad ins Nichts folgen. Abbildung 2 illustriert dieses Beispiel und zeigt den emergenten Effekt des Algorithmus.

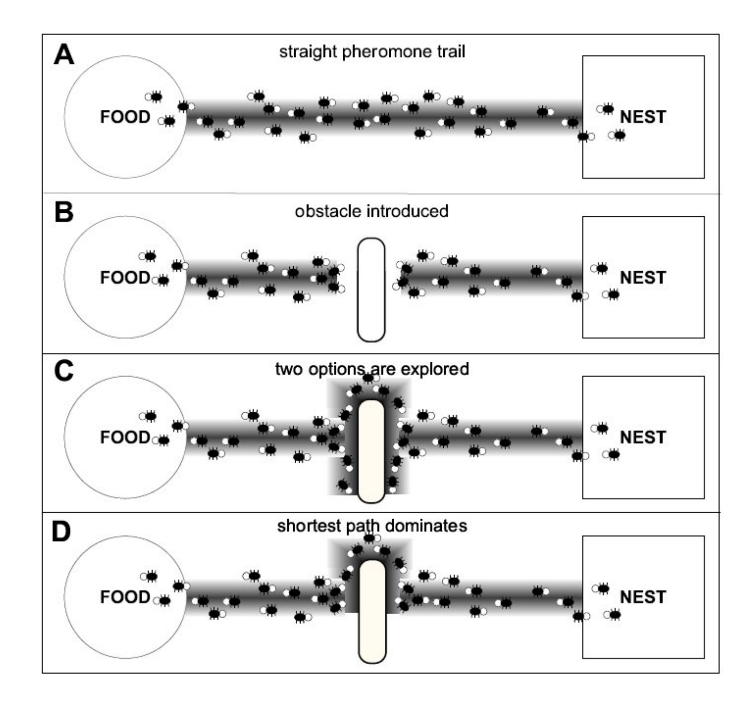


Abbildung 2. Beispiel eines Digitalen Pheromonpfad aus [3]

Nicht nur, dass mit dieser dezentralen Methode die Koordination der Ameisen zur Nahrung gewährleistet wird und eine Selbst-Organisation entsteht, es entwickelt sich auch eine Selbst-Optimierung des Systems. Werden z.B. von zwei verschiedenen Ameisen zwei verschiedene Wege vom Nest zur Nahrungsquelle gefunden, so wird über die Zeit der kürzere Pfad bevorzugt. Da die Ameisen auf jedem Rückweg von der Nahrung zum Nest Pheromone hinterlassen, wird dieser Duft auf der kürzeren der beiden Strecken intensiver als auf der längeren, da diese Strecke stärker frequentiert wird. Somit werden weitere Ameisen den kürzeren, stärker duftenden Weg bevorzugen, was wiederum zu einer erneuten Intensivierung des Pheromonpfades führt. Der kürzeste aller gefundenen Pfade, muss jedoch nicht unbedingt der kürzeste aller möglichen Pfad vom Nest zur Nahrungsquelle sein. Es ist durchaus denkbar, dass keine Ameise diesen, kürzesten aller möglichen Wege, gefunden hat.

Die Idee des Digitalen Pheromonpfades besteht darin, dieses effektive System der Natur auf die Informatik zu übertragen. Die Umgebung muss fähig sein, digitale Pheromone aufzunehmen, welche sich über die Zeit verflüchtigen. Die

Agenten sind sehr einfach strukturiert und müssen sobald sie ihr Ziel erreicht haben den Weg nach Hause finden, sowie Pheromone absetzen, bzw. diese in der Umgebung erkennen können. Die Richtung der Pheromone spielt dabei eine wichtige Rolle.

Eigenschaften des Digitalen Pheromonpfades:

- Durch das Zusammenspiel und die dezentrale Koordination aller Agenten untereinander organisiert sich das System selbst.
- Der Pfad stellt die Lösung des Problems im modellierten System dar.
- Das System erfährt eine Selbst-Optimierung, in dem kürzere Wege bevorzugt werden.
- Es werden nur lokale Informationen benötigt und nur die direkte Umgebung wird von den Agenten beeinflusst.
- Je nachdem wie die Objekte (Agenten, Umgebung, Pheromone) programmiert werden, kann man die Funktionsweise des Algorithmus beeinflussen.
  - Verschiedene Nahrungsquellen veranlassen die Agenten unterschiedliche Pheromone abzusetzen.
  - Wenn sich Pheromone schnell verflüchtigen, werden Wege zur Nahrung schnell vergessen und es werden neue Wege zu den Nahrungsquellen gefunden.
- Wichtige neue Informationen (Pfade zu neuen Nahrungsquellen) werden schnell integriert und überflüssige Informationen (Pfade zu aufgebrauchter Nahrung) werden schnell vergessen.
- Der Ausfall einzelner Agenten ist unkritisch, da sie nur einen sehr begrenzten Einfluss auf das gesamte System haben.
- Auch nach Änderungen der Umgebung finden die Agenten ihren Weg zum Ziel.
- Werden neue Pheromone auf einen bestehenden Pfad abgelegt, wirken diese als positives Feedback (siehe Abschnitt 2.6). Das Verflüchtigen der Pheromone hingegen wirkt als negatives Feedback.

Die Komplexität der Agenten und der Umgebung ist bei diesem Verfahren sehr gering gehalten, jedoch ist wie bei allen agentengesteuerten dezentralen Koordinationsmechanismen die Logik in den Agenten höher, als die der Umgebung. Agenten müssen unterschiedliche Pheromone erkennen, diese an die Umgebung abgeben können und sofern noch kein Pfad vorhanden ist, Nahrung suchen. Die Umgebung muss in diesem Modell fähig sein, Pheromone aufzunehmen. Um die Verflüchtigung bzw. das Verschwinden sollten sich die Pheromon Objekte selbst kümmern.

Einsatzgebiete sind z.B. laut [3]:

- Das Problem des Handlungsreisenden.
- Der Indizierungsalgorithums von Google.

– Mobiles Ad-Hoc Netzwerkmanagment.
– Die Kontrolle und Koordination von Schwärmen unbemannter militärischer Fahrzeuge zu Luft und/oder zu Land.

Den Unterschied zwischen den Verfahren Digitaler Pheromonpfad und Gradientenfeldern verdeutlicht Abbildung 3.

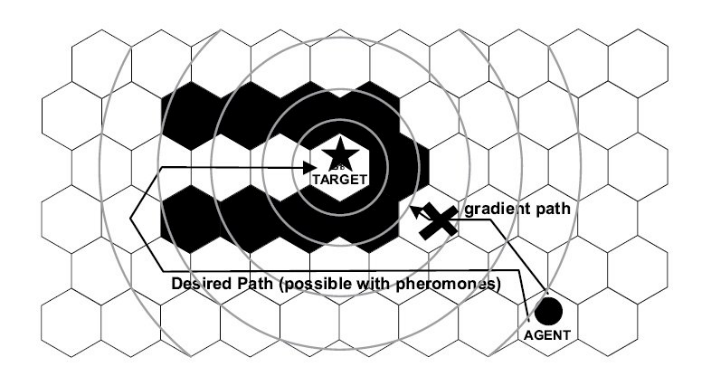


Abbildung 3. Gradientenfelder und Digitaler Pheromonpfade aus [3]

Über die Gradientenfelder würde der Agent den kürzesten Weg zur Quelle finden, dies ist beim Ameisenalgorithmus nicht zwingend der Fall. Bei diesem Beispiel ist jedoch zu erkennen, dass der Agent der auf die Verlaufsfelder reagiert, nicht zu seinem Ziel kommen kann, da ihn eine Mauer behindert. Ein Agent der den Digitalen Pheromonpfad Algorithmus verwendet, würde das Ziel irgendwann erreichen und auf dem Rückweg eine Pheromon Spur zum Ziel legen. Dadurch können nun andere Agenten diesem Pfad folgen und ihr Ziel erreichen.

Ein bedeutender Unterschied der beiden Verfahren ist, dass bei Gradientenfeldern die Umgebung den roten Teppich zur Verfügung stellt, die Koordination umgebungsgesteuert abläuft. Beim Digitalen Pheromonpfad müssen die Agenten selbst den roten Teppich erstellen. Es liegt eine agentengesteuerte dezentrale Koordination vor. Dies zeigt, dass unterschiedliche Einsatzgebiete und Anforderungen unterschiedliche dezentrale Koordinationsmechanismen erfordern.

4.3 Marktbasierte Kontrolle

Ein dezentrales Verfahren zur Koordination von Agenten, das auf einer Idee der Volkswirtschaftslehre basiert, ist die marktbasierte Kontrolle. Laut [4] ist diese Methode vor allem dann sinnvoll, wenn sich die Agenten knappe Ressourcen teilen müssen.

Die Grundprinzipien von Angebot und Nachfrage, sowie der daraus entstehende Marktpreis werden in Abbildung 4 verdeutlicht.

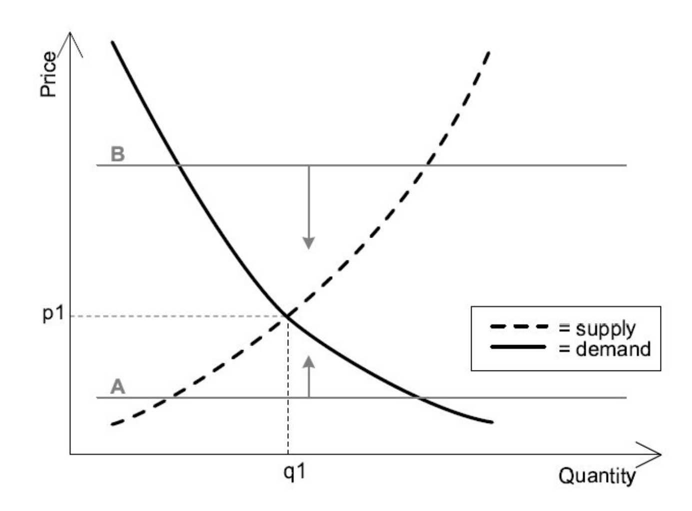


Abbildung 4. Angebot und Nachfrage und der Marktpreis aus [4]

Das Prinzip der Marktbasierten Kontrolle beruht auf der Nachfrage und dem jeweiligen Angebot von Gütern. Es gibt Käufer und Verkäufer die, unabhängig voneinander, eigennützig handeln. Güter werden verkauft, sobald ein für den Verkäufer akzeptabler Preis angeboten wird. Käufer werden so lange nach Gütern suchen, bis sie diese zu einem für sie akzeptablen Preis erhalten können. Wenn alle Teilnehmer des Systems ihren eigenen Interessen folgen, Verkäufer einen hohen Preis erzielen, Käufer möglichst wenig bezahlen wollen, wird sich ein Marktpreis einstellen. Dies ist der Preis, der zu einem Gleichgewicht aus Angebot und Nachfrage führt, daher auch Gleichgewichtspreis genannt wird. Wenn die Nachfrage für ein Gut steigt, wird auch der Preis für dieses Gut steigen, und umgekehrt. In Abbildung 4 kann man erkennen, dass im Fall A bei einem geringen Preis für das Gut, die Nachfrage hoch ist, das Angebot jedoch sehr niedrig, da wenige Anbieter zu diesem Preis verkaufen. Daher wird sich der Preis erhöhen, damit mehr Güter dieser Art angeboten werden, wodurch aber die Nachfrage für dieses Gut sinkt. Die Gerade verschiebt sich nach oben. Im Fall B ist der Preis sehr hoch, viele Verkäufer bieten ihre Güter zu diesem Preis an. Die Nachfrage ist deutlich geringer als das Angebot. Die Verkäufer werden ihre Preise senken, um mehr Güter abzusetzen. Mit fallenden Preisen steigt die Nachfrage und die Gerade verschiebt sich nach unten. In beiden Fällen wird der für Angebot und Nachfrage optimale Preis, der Marktpreis erreicht.

In [3] wird beschrieben, dass das Ziel der marktbasierten Kontrolle in der Informatik grundlegend verschieden zu den Theorien der Volkswirtschaftslehre ist. Die Prinzipien von Angebot und Nachfrage werden als gegeben hingenommen und es ist nicht die kritische Frage, ob diese Theorie das menschliche Verhalten reflektiert. Wichtig ist stattdessen, wie diese Theorie auf die Ressourcenvergabe

in Computersystemen übertragen werden kann. Die Agenten im System müssen wie Händler programmiert sein, die kaufen und verkaufen, um ihre benötigten Ressoucen zu erhalten. Das System ist selbst-organisierend, da sich der Gleichgewichtspreis einstellt.

Die Agenten des Systems handeln miteinander um die begrenzten Ressourcen des Systems. Dafür steht ihnen virtuelles Geld zur Verfügung, mit dem sie Angebote, für eine von ihnen benötigte Ressource, machen können. Sie werden versuchen, möglichst wenig für die benötigte Ressource zu zahlen. Agenten die diese Ressourcen besitzen bieten sie zu einem bestimmten Preis an und werden versuchen möglichst viel Geld zu erhalten. Jeder Agent im System ist eigennützig und entscheidet selbst, ob er momentan als Käufer oder Verkäufer agiert. Ob ein Agent für eine Ressource bietet, oder seine eigene zum Verkauf anbietet hängt von der lokalen Information des Agenten und dem aktuellen Marktpreis ab. Jeder Agent hat Preislimits, die sowohl für den Einkauf (als Einkäufer bezahlt er nicht mehr als das Preislimit vorgibt) als auch für den Verkauf (als Verkäufer verkauft er nicht unter diesem Preis) verschieden sind. Diese Preislimits sind jedoch nur lokal verfügbar und den anderen Agenten nicht bekannt. Wie im Modell der Volkswirtschaftslehre wird sich ein Gleichgewichtspreis einstellen, der ansteigt sobald die globale Nachfrage nach Ressourcen steigt. Ebenso wird er fallen, wenn die allgemeine Nachfrage nach Ressourcen fällt.

Wie schon im Abschnitt 4.1 CRL, kommunizieren die Agenten bei der marktbasierten Kontrolle direkt miteinander. Die Angebote werden direkt an einen Agenten in der Nachbarschaft gemacht und laufen nicht wie beim Digitalen Pheromonpfad aus Abschnitt 4.2 über die Umgebung. Auch wird die Koordination der einzelnen Agenten nicht von der Umgebung vorgenommen. Die Agenten reagieren unabhängig, nach ihren lokalen Bedürfnissen. Bei diesem Verfahren kann über allgemeine Vorgaben das System beeinflusst werden. So kann das Preislimit, das jeder Agent hat, die Anzahl der Agenten, die Anzahl der virtuellen Geldeinheiten die jeder Agent zur Verfügung hat und die Anzahl der zur Verfügung stehenden Ressourcen das System verändern.

Eigenschaften der Marktbasierten Kontrolle:

- Der Markt liefert den Agenten Informationen über den Ressourcenbedarf. Ein hoher Marktpreis geht mit einem aktuell hohen Ressourcenbedarf oder einem niedrigen Ressourcenangebot einher.
- Der Preis agiert als Feedback in diesem System. Ein hoher Preis veranlasst Agenten weniger zu kaufen. Aufgrund des negativen Feedbacks, der geringen Nachfrage, wird der Preis fallen. Bei niedrigen Preisen ist es umgekehrt, es tritt somit das postive Feedback auf, es wird mehr gekauft. Durch die Kombination von negativem und postivem Feedback nähert sich das System dem Gleichgewichtspreis an.
- Werden einzelnen Agenten zu Beginn weniger virtuelle Geldeinheiten zur Verfügung gestellt, kann somit eine geringere Wichtigkeit des Agenten modelliert werden.

- Dieses Modell führt laut [4] zu einer Pareto-Optimierung des Systems. Dies bedeutet laut [11], dass kein Beteiligter besser gestellt werden kann, ohne einen anderen schlechter zu stellen. Für diesen Algorithmus bedeutet das, dass nicht zwingend eine globale, optimale Lösung gefunden wurde, denn durch die Schlechterstellung eines unwichtigeren Agenten eine Besserstellung eines wichtigeren Agenten möglich wäre, was das System optimieren könnte.

Von allen hier vorgestellten Algorithmen ist die Komplexität der Agenten in diesem System am größten. Sie müssen eine Logik enthalten, die vorgibt, wann Ressourcen gekauft werden, bzw. zu welchem Preis verkauft wird. Dies hängt immer vom aktuellen Marktpreis und den lokalen Anforderungen des jeweiligen Agenten ab. Die Komplexität der einzelnen Agenten ist jedoch klein im Vergleich zur Komplexität, wenn ein zentral organisiertes Gesamtsystem programmiert werden müsste. An die Umgebung der Agenten wird keine besondere Anforderung gestellt.

Als Anwendungen wird in [3] beschrieben, dass die Ressourcen Herstellungsmaschinen, die Bandbreite in einem Netzwerk oder Arbeitsaufgaben (Tasks) sein können.

5 Koordinationsmechansimen und realisierte Selbst-x Eigenschaften

Die vorgestellten Methoden zur dezentralen Koordination realisieren verschiedene Selbst-x Eigenschaften. Die nun folgende Auflistung zeigt auf einen Blick, welcher Koordinationsmechanismus welche Selbst-x Eigenschaften mit sich bringt.

Selbst-Optimierung und Selbst-Konfiguration entsteht in jedem der vorgestellten Gesamtsysteme. Bei den Gradientenfeldern, wie auch beim digitalen Pheromonpfad, zeigt sich die Selbst-Konfiguration dadurch, dass jeder Agent sein Ziel erreicht, ohne dass ihm explizit einprogrammiert wurde, wie er dieses Ziel erreichen soll. Die Agenten eines CRL Systems konfigurieren sich, indem sie von einander lernen und dadurch zu einer Selbst-Optimierung des System gelangen. Bei der marktbasierten Kontrolle zeigt sich die Konfiguration dadurch, dass sich automatisch der Gleichgewichtspreis einstellt.

Bei der Selbst-Optimierung des digitalen Pheromonpfades muss, wie in Abschnitt 4.2 erläutert, die Einschränkung gemacht werden, dass der gefunden Pfad nicht der optimalste, also kürzeste, vom Nest zu Nahrung sein muss. Es wird nur der optimalste aller gefundenen Pfade bevorzugt. Auch bei der markbasierten Kontrolle entsteht wie in Abschnitt 4.3 schon erläutert, lediglich eine Pareto-Optimierung des Systems. Bei CRL hingegen, wird die optimalste Lösung für ein DOP gewählt und bei den Gradientenfeldern der direkte, also kürzeste Weg zur Quelle. Hier kann es, wie am Ende von Abschnitt 4.2 gezeigt, zu Problemen kommen, wenn sich ein Agent gierig auf eine Quelle zu beweget.

Selbst-Heilung bei CRL zeigt sich dadurch, dass schlechte Lösungsansätze erkannt werden, diese für die Problemlösung nicht in Betracht gezogen werden, sondern optimalere Lösungen angewendet werden. Beim digitalen Pheromonpfad

bedeutete Selbst-Heilung, dass die Agenten des Systems Pfade, die nicht mehr zur Nahrung führen, nicht weiter verwenden und nach einer bestimmten Zeit diese Duftpfade verschwinden. Bei der marktbasierten Kontrolle stellt sich nach dem Ausfall einiger Agenten immer wieder der Gleichgewichtspreis ein. Das System heilt sich selbst, da die Ressourcenverteilung, auch nach dem Ausfall einiger Agenten, wieder Pareto-Optimiert wird.

Selbst-Schutz bedeutet im Fall von CRL, dass Agenten, die schlechte Lösungsansätze einbringen wollen, nicht berücksichtigt werden. Wenn diese Agenten nicht von ihren Nachbarn lernen wollen, werden sie nicht am System teilnehmen können, da keiner Ihrer Nachbarn diese schlechtere Lösung für sein DOP verwenden wird. Beim digitalen Pheromonpfad schützt sich das System vor längeren Wegen dadurch, dass es wenigen Agenten nie möglich sein wird, das Gesamtsystem so zu beeinflussen, dass ein längerer Weg bevorzugt wird. Bei der marktbasierten Kontrolle schütz sich das System vor Wucher, indem mit diesen Agenten nie ein Handel zustande kommen wird.

Tabelle 1. Koordinationsmechansimen und realisierte Selbst-x Eigenschaften

Koordinationsmechansimus	realisierte Selbst-x Eigenschaften
Gradientenfelder	- Selbst-Konfiguration - Selbst-Optimierung
CRL	- Selbst-Konfiguration - Selbst-Optimierung - Selbst-Heilung - Selbst-Schutz
Digitaler Pheromonpfad	- Selbst-Konfiguration - Selbst-Optimierung - Selbst-Heilung - Selbst-Schutz
Marktbasierte Kontrolle	- Selbst-Konfiguration - Selbst-Optimierung (Pareto-Optimiert) - Selbst-Heilung - Selbst-Schutz

6 Schluss

Neben den in dieser Arbeit erläuterten dezentralen Koordinationsmechanismen sind vor allem die in [3] beschriebenen Verfahren Tags und Tokens zu erwähnen.

Das Tag-Verfahren basiert auf einem sozialen Grundgedanken. Soziale Strukturen bilden sich spontan und ergeben zweckmäßige Strukturen, Institutionen und Organisationen. Aufbauend auf dieser Idee schließen sich beim Tag Verfahren Agenten zu Gruppen zusammen, wenn Sie die selben Interessen haben.

Ob Agenten die selben Interessen haben, können sie an den Tags der anderen erkennen. Vergleichbar sind diese Tags z.B. mit dem Kleidungsstil verschiedener sozialer Gruppierungen. Als Mensch werde ich die Zugehörigkeit zu einer sozialen Gruppe zeigen, indem ich mich ihr optisch anpasse. Analog dazu kopieren Agenten die Tags der jeweiligen Gruppe, um ihre Zugehörigkeit zu demonstrieren. Anwendungsgebiete sind laut [3] z.B. das Peer-to-Peer Filesharing, bei dem sich Gruppen bilden, welche die gleichen Interessen haben und Teilnehmer, die keine Dateien anbieten, sondern nur herunterladen möchten, ausgeschlossen werden. Das Gesamtsystem wird zum Erfolg, wenn alle Agenten nicht selbstsüchtig sondern im Sinne der Gruppe handeln.

Die Koordination mittels Tokens wird laut [3] vorallem dann verwendet, wenn es um den Zugriff auf begrenzte Ressourcen geht. Agenten erhalten das Recht auf die Ressourcen zuzugreifen, wenn Sie das Token besitzen. Sobald Sie die Resource nicht mehr benötigen wird der Token weitergegeben und der nächste Agent kann auf die Resource zugreifen. Dieses Verfahren kann z.B. mit einer Karte in einer Firma verglichen werden, die dem Besitzer die Möglichkeit gibt den Kopierer zu benutzen. Sobald ein Mitarbeiter den Kopierer nicht mehr benötigt, gibt er die Karte an einen Kollegen weiter. Dieser hat nun aufgrund der Karte die Möglichkeit die Resource, den Kopierer, zu verwenden.

Alle in dieser Arbeit beschriebenen Verfahren beruhen auf einer Grundidee aus der Natur oder aus dem sozialen Gefüge unserer Gemeinschaft. Es sind einfache Verfahren, die dadurch, dass sie auf mehrere Agenten angewendet werden, einen größeren Effekt erzielen, als man es den einzelnen Agenten einprogrammieren könnte. Meiner Meinung nach liegt darin die Zukunft, denn nicht immer komplexere Systeme können die Probleme lösen, da man durch die Komplexität schnell an die Grenzen der menschlichen Überschaubarkeit stößt. Dass das Zusammenspiel von mehreren kleinen, einfachen Komponenten uns in der Softwareentwicklung weiter bringen kann, zeigt sich schon heute im Hardwarebereich. Bei Multikern-Prozessoren arbeiten mehrere, einfache Komponenten zusammen und sind leistungsfähiger, als eine komplexe, Ein-Kern Prozessor Komponente.

Literatur

[1] S. Apel and E. Buchmann. Biology inspired optimizations of peer-to-peer overlay networks, December 2005.

[2] T. de Wolf. Analysing and engineering self-organising emergent applications, May 2007.

[3] T. de Wolf and T. Holvoet. A catalogue of decentralised coordination mechansims for designing self-organising emergent applications, August 2006.

[4] T. de Wolf and T. Holvoet. Decentralised coordination mechanisms as design patterns for self-organising emergent applications., 2006.

[5] J. Dowling. The decentralised coordination of self-adaptive components for autonomic distributed systems, October 2004.

[6] J. Dowling and V. Cahill. Self-managed decentralised systems using k-components and collaborative reinforcement learning, October 2004.

[7] M. Encarta. de.encarta.msn.com/Lernen_durch_Versuch_und_Irrtum.html.

[8] J. Kennedy and R. C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann, 2001.

[9] M. Lexikon. lexikon.meyers.de/meyers/Emergenz.

[10] J. A. McCann and M. Huebscher. Evaluation issues in autonomic computing, October 2004.

[11] W. M. Verlag. wissen.spiegel.de/wissen/dokument.html?id=54310500.

Design Patterns für autonome dezentrale Systeme

Martin Burkhard

Universität Augsburg
martin.burkhard@web.de

Zusammenfassung Diese Arbeit beschäftigt sich mit den Design Patterns für autonome dezentrale Systeme. Zunächst werden Entwurfsmuster aus der objektorientierten Programmierung näher betrachtet und auf ein Klassifizierungsschema eingegangen. Im Zentrum der Untersuchung steht die Frage nach der Notwendigkeit von Design Patterns für autonome dezentrale Systeme. Abschließend stellt dieses Dokument eine Auswahl an Entwurfsmuster vor und veranschaulicht deren Anwendungsmöglichkeit am Fallbeispiel eines Paketlieferdienstes.

1 Einleitung

Der Begriff *Pattern* (Muster) wurde 1977 vom Architekturprofessor Christopher Alexander in seinem Buch *A Pattern Language - Towns-Building-Construction* [1] geprägt:

> "Each Pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Demnach beschreibt ein Pattern ein wiederkehrendes Problem und bietet für dieses Problem eine geeignete allgemeingültige Lösungsmöglichkeit in Form einer Vorlage an, die sich jederzeit wiederholt anwenden lässt.

Inzwischen haben sich *Design Patterns* (Entwurfsmuster) als Lösungsstrukturen für Entwurfsprobleme in der Praxis bewährt, indem sie Erfahrungswissen von Experten leichter zugänglich machen. Entwickler profitieren demnach durch die Formalisierung und Dokumentation von erprobten Entwürfen, indem sich Patterns auf wiederkehrende Probleme effizient anwenden lassen.

In dieser Arbeit werden Design Patterns für *autonome dezentrale Systeme* untersucht. Dazu werden im Folgenden die Konzepte und Prinzipien der Patterns aus der objektorientierten Programmierung betrachtet, um festzustellen, welchen Nutzen spezielle Patterns für autonome dezentrale Systeme haben. Abschließend wird auf eine Auswahl an existierenden Design Patterns für autonome dezentrale Systeme eingegangen und diese in einem Fallbeispiel angewendet.

In Kapitel 2 wird eine Übersicht über objektorientierte Entwurfsmuster präsentiert. Darin wird näher auf die Design Patterns von Erich Gamma et al. eingegangen und eine klassische Vorlage für die Beschreibung der Design Patterns vorgestellt.

Kapitel 3 beschäftigt sich mit den Design Patterns für autonome dezentrale Systeme und zieht den Vergleich zu den Entwurfsmustern der objektorientierten Programmierung. Zunächst wird die verwendete Terminologie definiert und der Zusammenhang zwischen autonomen dezentralen Systemen und Multi-Agenten-Systemen hergestellt, bevor anschließend auf Patterns für autonome dezentrale Systeme eingegangen wird.

Abschließend kommen im 4. Kapitel die vorgestellten Design Patterns im Anwendungsbeispiel eines Paketlieferdienstes zum Einsatz.

2 Objektorientierte Patterns

Auf ihrer Suche nach einer neuen Methodik für die objektorientierte Programmierung (OOP) übertrugen Kent Beck und Ward Cunningham 1987 das Konzept des Pattern von Alexander auf die Softwaretechnik, um den neuen Herausforderungen von OOP zu begegnen und das Potenzial der OOP besser auszuschöpfen. [8]

Inspiriert von dieser Idee ist eine Reihe unterschiedlicher Design Patterns entstanden und Kataloge, welche eine Vielzahl der Entwurfsmuster zusammenfassen. Zwei bekannte Entwurfsmuster-Kataloge, die Design Pattern der *Gang-of-Four* (GoF) und die *General Responsibility Assignment Software Patterns* (GRASP), werden in folgenden Abschnitten vorgestellt. Weitere bekannte Werke zum Thema Design Patterns für objektorientierte Softwareentwicklung sind *Design Patterns for Object-Oriented Software Development* [21] von Wolfgang Pree und *Patterns of enterprise application architecture* [12] von Martin Fowler.

Maßgeblich für den Einsatz von Entwurfsmuster ist dabei die Einteilung (Klassifikation) der Design Patterns nach gemeinsamen Merkmalen. Das Klassifizierungsschema erleichtert dabei die Identifikation der Lösungsmöglichkeit anhand der Problembeschreibung und ermöglicht den Vergleich zwischen den Patterns untereinander.

2.1 Klassifizierungsschema

Bei der Beschreibung der Design Patterns entscheidet sich der Autor, neben übergeordneter Kategorie, für eine konsistente, einheitliche und strukturelle Formatvorlage, die allen Entwufsmustern zugrunde gelegt wird. Dieses Klassifizierungsschema gibt dem Softwarearchitekten nicht nur die Möglichkeit für ihre Problemstellung die passenden Lösungen zu finden, sondern die Alternativen gegenüberzustellen, auszuwählen und anzuwenden.

Ein klassisches Beispiel einer solchen Vorlage ist das Klassifizierungsschema aus dem Buch der GoF [14]:

Pattern Name and Classification - der Name des Patterns muss ausdrucksstark genug sein, um das Wesentliche des Patterns zu beschreiben. Das Pattern wird darüber hinaus einer geeigneten übergeordneten Kategorie zugeteilt.

Intent - es stellt sich die Frage, welchem Zweck das Pattern dient? Welches Designproblem behandelt das Pattern?

Also Known As - eine Liste an Synonyme für das Entwurfsmuster.

Motivation - ein Szenario soll das Designproblem verdeutlichen sowie herausstellen, auf welche Weise das Pattern dieses Problem löst.

Applicability - bei welcher Gegebenheit kann das Entwurfsmuster angewendet werden? Welche Beispiele existieren für schlechtes Design? Wie können diese Gegebenheiten erkannt werden?

Structure - zur Beschreibung der Struktur kommt als grafische Notation, die *Object Modeling Technique* (OMT) [23] von James Rumbaugh zum Einsatz, aus der wenige Jahre später die Unified Modeling Language (UML) [6, 11] entstanden ist.

Participants - führt die am Entwurfsmuster beteiligten Klassen auf.

Collaborations - beschreibt die Zusammenarbeit der beiligten Klassen.

Consequences - bietet eine Übersicht über die Vor- und Nachteile, die beim Einsatz des Patterns entstehen.

Implementation - eine Reihe von Tipps, Hinweisen und Techniken sollen den Einsatz erleichtern und die falsche Anwendung vermeiden.

Sample Code - durch Programmfragmente wird die Implementierung beispielhaft dargestellt.

Known Uses - schildert Anwendungsgebiete aus der Praxis.

Related Patterns - zeigt die Verwandschaftsbeziehungen zwischen den Entwurfsmustern auf.

Eine ausdrucksvolle Bezeichnung der Pattern hilft Softwareingenieuren sich über die jeweiligen Entwurfsmuster zu verständigen. Im Mittelpunkt des Schemas stehen die Problembeschreibung und die Lösungsmöglichkeit, welche den Kern des Problems treffen müssen, aber gleichzeitig allgemein genug verfasst sein sollten. Ein Szenario sowie eine visuelle Beschreibung der Struktur unterstützen das Verständnis der Problembeschreibung. Ein Beispielprogramm zeigt zudem, wie sich die Lösung exemplarisch verwirklichen lässt.

Neben dem Klassifizierungsschema ist in Design Pattern Katalogen oftmals eine *Problem-/Lösungstabelle* aufgeführt, die einen Gesamtüberblick über alle vorhandenen Patterns bietet. Eine Kurzbeschreibung der Problemstellung und der zugehörigen Lösungsmöglichkeit erleichtert die Auswahl der Entwurfsmuster für den späteren Entwicklungseinsatz.

2.2 Gang-of-Four Design Patterns

Im Jahr 1994 erschien *Design Patterns - Elements of Resusable Object-Oriented Software* [14] von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, der sogenannten *Gang-of-Four* (GoF). In diesem Buch werden insgesamt 23 Entwurfsmuster katalogisiert, welche für die objektorientierte Softwareentwicklung eingesetzt werden können.

Die Autoren dieses Buches haben darauf Wert gelegt, dass bewährte Verfahren (Best Practices) aus der objektorientierten Softwareentwicklung beschrieben werden, welche den Entwurf und die Umsetzung von Software-Architekturen und -Komponenten erleichtern. Weiter verfolgen sie generelle Ansätze für den Softwareentwurf und verzichten auf domänen-spezifische Patterns.

Auf diese Weise profitieren unerfahrene Softwareentwickler direkt vom Wissen der Sachverständigen, um Code sauber zu strukturieren und wiederkehrende Probleme bei der Softwareerstellung einheitlich zu lösen. [22]

Die GoF unterscheiden anhand des Einsatzgebietes der Design Patterns grundlegend zwischen *Creational Pattern*, *Structural Pattern* und *Behavioral Pattern*.

Die Creational Pattern (Erzeugungsmuster) widmen sich dem Objekterzeugungsprozess. Das Ziel ist die Erzeugung des Objekts von seiner Repräsentation zu entkoppeln, um die Unabhängigkeit und dadurch höhere Flexibilität zu erreichen. Bekannte Vertreter sind das Singleton Pattern, das Factory Pattern und das Prototype Pattern.

Die Structural Pattern (Strukturmuster) zeigen die Möglichkeiten zur Anordnung und Zusammensetzung von Objekten und Klassen auf, um größere Strukturen zu erzeugen. Bekannte Patterns sind Decorator, Facade und Proxy.

Bei den Behavioral Pattern (Verhaltensmuster) steht die Objektinteraktion im Mittelpunkt. Spezielle Kommunikations-Patterns sollen eine flexible Kommunikation zwischen den beteiligten Objekten ermöglichen. Beispiele für Behavioral Patterns sind das Mediator Pattern, Observer Pattern und Strategy Pattern.

Zwei Pattern [14] mit jeweils unterschiedlicher Kategorie werden in einem verkürzten Schema vorgestellt:

Singleton (Creational Pattern)

Intent

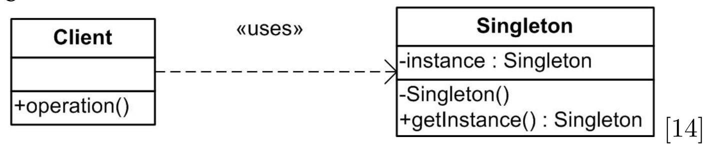
Stellt sicher, dass von einer Klasse lediglich eine Instanz erzeugt werden kann und globaler Zugriff darauf besteht.

Motivation

(1) Lediglich ein Dateisystem oder ein Window-Manager soll existieren.
(2) Über globale Variablen wird zwar der Zugriff sichergesteellt, aber nicht die mehrfache Instanziierung verhindert.

Applicability
(1) Für den Fall, dass genau eine Instanz einer Klasse erstellt werden darf.
(2) Falls die Klasse mithilfe von Vererbung erweiterbar sein soll und eine erweiterte Klasse ohne Änderung nutzbar sein soll.

Structure



[14]

Participants
(1) Singleton - definiert eine Methode, über welche auf die Instanz zugegriffen werden kann und verhindert, dass unkontrolliert eine Instanz der Klasse erzeugt wird.
(2) Client - greift auf das Singleton zu.

Consequences
(1) Der Zugriff auf die Instanz erfolgt kontrolliert.
(2) Die Anzahl der erstellbaren Instanzen ist festlegbar.
(3) Durch Vererbung ist Erweiterung um Attribute und Methoden möglich.

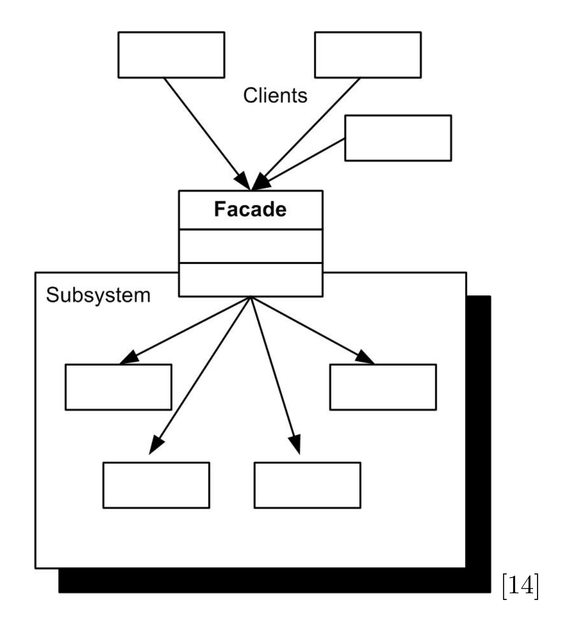
Facade (Structural Pattern)

Intent
(1) Stellt eine einfache standardisierte Schnittstelle zu einem Subsystem dar.
(2) Ermöglicht über die Schnittstelle den Zugriff auf die Funktionalität des Subsystems.

Motivation
(1) Starke Kopplung zwischen Clients und Subsystem.
(2) Direkte Aufrufe sind aufwendig oder fehleranfällig.

Applicability
(1) Eine vereinfachte Sicht soll auf ein komplexes Subsystem zur Verfügung gestellt werden.
(2) Viele Abhängigkeiten bestehen zwischen Client-Klassen und den Subsystem-Klassen.

Structure



[14]

Participants

(1) Facade - Kennt die Subsystem-Klassen und leitet Anfragen an sie weiter
(2) Subsystem-Klassen - Implementieren die Funktionalität, kennen aber die Facade nicht.
(3) Client-Klassen - greifen über die Facade auf das Subsystem zu.

Consequences

(1) Entkoppelt die Client-Klassen vom Subsystem.
(2) Clients und Subsystem können unabhängig voneinander implementiert werden.
(3) Zusätzlicher Overhead bei einfachem Subsystem.

Die allgemein beschriebenen Patterns der Gang-of-Four können nicht alle Problemfälle abdecken. Aus diesem Grund wählt ein Softwarearchitekt zunächst die Patterns, welche die aktuelle Problemsituation am besten beschreiben und passt die Patterns entsprechend der Problemstellung an.

2.3 GRASP Patterns

Neben den bekannten Design Pattern der Gang-of-Four existieren weitere Vorgehensweisen und Regeln wie beispielsweise die *General Responsibility Assignment Software Patterns* (GRASP). Craig Larman beschreibt in seinem Buch *Applying UML and patterns* [17] die GRASP Pattern als eine Art Lernhilfe, um die fundamentalen Prinzipien des objektorientierten Designs (OOD) zu vermitteln. Insgesamt neun GRASP Pattern sollen dem Softwarearchitekten helfen, die Zuständigkeiten von Objekten logisch und systematisch zu bestimmen.

Mit Information Expert, High Cohesion und Controller werden drei der GRASP Pattern vorgestellt:

Information Expert

Intent Nach welchem soll die Zuständigkeit vergeben werden?
Solution Die Klasse ist zuständig, welche die relevante Information besitzt.
Consequences Analogie aus der realen Welt.

High Cohesion

Intent Wie kann Komplexität reduziert werden?
Solution Weise Klassen nur möglichst stark zusammenhängende Zuständigkeiten zu.
Consequences Widerspruch zum GRASP Pattern *Low Coupling* möglich.

Controller

Intent Wer ist für die Entgegennahme von Systemevents zuständig?
Solution Die Zuständigkeit wird an eine Facade-Klasse oder eine Use-Case-Klasse vergeben.
Consequences Der Zugriff erfolgt über eine einheitliche Schnittstelle, die vom Subsystem abstrahiert.

Die vorgestellten Patterns lassen bereits erkennen, dass GRASP vorwiegend eine Zusammenfassung von grundlegenden Richtlinien darstellt, die bei der objektorientierten Analyse angewendet werden. Während die GoF Pattern konkret auf die Implementierung eingehen, beschränken sich die GRASP Pattern auf textuelle Empfehlungen. Im Folgenden Abschnitt 3 sollen Entwurfsmuster für autonome dezentrale Systeme untersucht werden und den GoF und GRASP Pattern gegenübergestellt werden.

3 Design Patterns für autonome dezentrale Systeme

Der Begriff *autonom* stammt aus dem Griechischen und bedeutet selbstständig oder auch unabhängig [7]. Ein autonomes System soll demnach selbstständig und unabhängig handeln können.

Unter einem *dezentralen System* werden Systeme verstanden, denen eine hierarchisch übergeordnete, koordinierende Instanz fehlt. Die Koordination beruht in diesem Fall auf der Verhandlung zwischen gleichrangigen Einheiten.

Ein *autonomes dezentrales System* ist demnach selbst für seine Koordination und die Beschaffung benötigter Ressourcen verantwortlich, um seine Aufgaben zu erfüllen. Das System benötigt zu diesem Zweck eine geeignete Umgebung, über welche sie mit anderen Mitgliedern dieser Umgebung kommunizieren und interagieren kann. In diesem Zusammenhang wird auch von dezentralen *selbstorganisierenden Systemen* gesprochen. Selbstorganisation ist nach [5]:

> ”das spontane Entstehen von (neuen) räuml. und zeitl. Strukturen in dynam. Systemen, das auf das kooperative Wirken von Teilsystemen zurückgeht. [...] Die Selbstorganisation erweist sich dabei als Vorgang zur Entstehung von Ordnung und Komplexität aus einem System selbst heraus. [...]”

Innerhalb selbstorganisierender Systeme arbeiten demzufolge autonome Teilsysteme kooperativ zusammen, um komplexere Aufgaben zu bewältigen. Oftmals sind die einzelnen Teilsysteme nicht in der Lage ihre Ziele ohne die Unterstützung anderer Teilsysteme zu erreichen oder zumindest nicht mit der gleichen Effizienz.

Im weiteren Verlauf soll untersucht werden, ob Design Patterns für autonome dezentrale Systeme existieren, wie sie aufgebaut sind und wofür diese Patterns angewendet werden können. Eine mögliche Umsetzung dezentral organisierter Systeme sind *Multi-Agenten-Systeme*, die im folgenden Abschnitt 3.1 vorgestellt werden.

3.1 Agenten-orientiertes Softwareengineering

Im Softwaretechnik-Bereich kann auf bewährte Konzepte der OOP zurückgegriffen werden und die Unified Modeling Language (UML) ist eine weitverbreitete und standardisierte Sprache, um objektorientierte Systeme konzeptionell zu entwerfen. Dagegen besteht für agentenorientiertes Softwareengineering (AOSE) kein Konsens über Begriffe, Vorgehensweisen und Konzepte. Folglich existieren verschiedene, oftmals proprietäre Methodologien[1] mit jeweils eigenen Definitionen, Konzepten und Metamodellen, die lediglich für spezifische Aufgabengebiete einsetzbar sind [4].

Das zentrale Element des agentenorientierten Softwareengineering (AOSE) ist der Begriff des *Agenten*. Eine Reihe von Artikeln [13, 27, 28] setzen sich detailliert mit den Konzepten des Agenten, *Software-Agenten* und der *Multi-Agenten-Systeme* auseinander und versuchen die Begriffe treffend zu umschreiben und abzugrenzen. Dennoch bestehen weiterhin große Meinungsverschiedenheiten über die zentralen Eigenschaften eines Agenten. Nach [25] stellt die folgende Beschreibung die nützlichste und am weitesten akzeptierteste Variante dar:

> "An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives." [28]

Demnach verbirgt sich hinter einem Agenten ein System, das sich in einem gewissen Umfeld befindet und welches die Fähigkeit besitzt, flexible und autonome Handlungen innerhalb dieses Umfeldes nachzugehen, um seine zur Design-Zeit festgelegten Ziele zu erfüllen.

Während die Definition lediglich auf einen einzelnen Agenten eingeht, ist für diese Arbeit der Begriff des *Multi-Agenten-Systems* von zentraler Bedeutung. Bei einem Multi-Agenten-System handelt es sich um eine Menge von einzelnen Agenten, die nicht nur autonom handeln, sondern auch miteinander interagieren können, um ein gemeinsames Ziel zu erreichen.

[1] "Methodologie [griechisch] die, Lehre von den in den Einzelwissenschaften angewendeten Methoden, als Teil der Logik zentraler Gegenstandsbereich der gegenwärtigen Wissenschaftstheorie." [7]

Für die weitere Untersuchung von autonomen dezentralen Systemen liegt der Fokus auf den Multi-Agenten-Systemen, die über eigene Entwicklungsmethodologien verfügen.

3.2 Entwicklungsmethodologien

Eine Entwicklungsmethodologie besteht üblicherweise aus einem Software-Entwicklungsprozess und einer Modellierungssprache. Der Entwicklungsprozess legt unter anderem das Vorgehensmodell, die Prozess-Phasen sowie den auszuführenden Aktivitäten fest. Beispiele für Vorgehensmodelle sind das Wasserfallmodell, V-Modell, Spiralmodell und der Rational Unified Process. Mithilfe der Modellierungssprache werden die Modelle auf Basis einer grafischen Notation (Syntax) erstellt. [15]

Bevor ein Softwarearchitekt mit der Analyse, Design und Implementierung beginnen kann, muss er sich eine Übersicht verschaffen, welche Technologien verfügbar sind und mit welchen Vorgehensweisen, Architekturen, Sprachen und Werkzeugen er sein Ziel erreichen kann.

Im Falle der Multi-Agenten-Systeme existieren bereits eine Reihe von Methodologien und Konzepte, auf die bei der Entwicklung von autonomen dezentralen Systemen zurückgegriffen werden können.

Tabelle 1 bietet eine Übersicht über die bekanntesten Entwicklungsmethodologien, die in folgende drei Kategorien eingeordnet wurden:

Knowledge Engineering - der Schwerpunkt liegt auf der Identifikation, Erfassung und Modellierung von Wissen, welches von Agenten verarbeitet wird.

Agenten-orientierte Ansätze - betrachtet werden soziale Hierarchien wie z.B. Agent, Gruppe oder Organisation.

Objektorientierte Ansätze - die bestehenden Konzepte aus der OOP werden um Agenten-Technologie erweitert.

Knowledge Engineering	Agenten-orientierte Ansätze	Objektorientierte Ansätze
CoMoMAS	Gaia	KGR
MAS-CommonKADS	SODA	MaSE
CommonKADS	Cassiopeia	MASSIVE
MIKE	AALAADIN	AOAD
PROTEGE	ROADMAP	MASB
	ADELFE	AOR
	EXPAND	PASSI
		Prometheus
		Tropos
		MESSAGE

Tabelle 1. Übersicht über die Entwicklungsmethodologien [15, 25].

Speziell für die Entwicklung von agenten-basierten Systemen ist bereits eine große Anzahl von agenten-spezifischen Sprachen verfügbar, die in drei Hauptkategorien eingeteilt werden können: *agenten-orientierte, interaktions-orientierte und markt-orientierte Programmierung.* Softwareingenieure können mithilfe dieser proprietären Sprachen die Anforderungen von agenten-basierten Systemen besser realisieren. Eine gute Übersicht zu den agenten-spezifischen Sprachen bietet [25].

Eine wichtige Erkenntnis bei der Untersuchung der Design Patterns ist, dass ein Pattern von den Modellen abhängig ist, die in der angewandten Methodologie verwendet werden. Die Modelle und deren Konzepte werden wiederum durch die Metamodelle beschrieben.

3.3 Metamodelle

Ein Metamodell definiert die Sprache zur Beschreibung der Modelle. Es umfasst die Konzepte, Syntax und Semantik, die bei der Modellierung angewendet werden. Im Falle der UML sind das beispielsweise *Klassen, Attribute, Operationen* und *Komponenten*, wie sie aus der OOP bekannt sind. [3]

In den im vorherigen Abschnitt vorgestellten Entwicklungsmethodologien werden anstelle der UML vorwiegend domänenspezifische Metamodelle zur Beschreibung der Konzepte der Multi-Agenten-Systeme eingesetzt. Die MAS-Modelle besitzen deshalb auch eine andere Semantik, als sie beispielsweise von der UML definiert wird. Aufgrund des unterschiedlichen Sinngehalts der Modelle und Konzepte lassen sich die bestehenden Design Patterns der OOP nicht immer auf autonome dezentrale Systeme bzw. Multi-Agenten-Systeme übertragen. Stattdessen müssen für die domänenspezifischen Konzepte jeweils eigene Design Patterns erstellt werden.

Wie bereits die Übersicht in Tabelle 1 zeigt, können dezentrale autonome Systeme auch mit objektorientierten Sprache wie z.B. Java, C++ oder C# realisiert werden, unter Verwendung der vorgestellten GoF und GRASP Patterns. Bei der Umsetzung der Konzepte und Mechanismen (Emergenz[2], Stigmergie[3], etc.) der dezentralen autonomen Systeme wird dann von der objektorientieren Implementierungsebene abstrahiert. Oftmals werden auf dieser Abstraktionsebene die besprochenen Metamodelle und Programmiersprachen eingesetzt, die auf die Paradigmen und Konzepte der dezentralen autonomen Systeme beruhen.

Demnach können die objektorientieren Entwurfsmuster und die Design Patterns der dezentralen autonomen Systeme innerhalb der selben Architektur auf verschiedenen Abstraktionsebenen verwendet werden.

Um die Abhängigkeit der Design Patterns von der jeweiligen Methodologie zu lösen, haben Carole Bernon et al. in ihrer Studie zwischen ADELFE, Gaia und PASSI einen Vergleich angestellt. Sie haben dabei nicht nur die Konzepte der jeweiligen Metamodelle analysiert, sondern versucht aus den vielversprechendsten

[2] "Emergenz ist dabei als die Eigenschaft eines Gesamtsystems definiert, welches nicht durch einfache Summation von Teileigenschaften errechnet werden kann." [19]

[3] Bei Stigmergie handelt es sich um einen speziellen Selbstorganisationsmechanismus, der bei der Kommunikation die Umgebung mit einbezieht. Siehe [15]

Aspekten der einzelnen Metamodelle ein einheitliches Metamodell zu erstellen. Jedoch war das Metamodell zu umfangreich, um darauf eine übergreifende Entwicklungsmethodologie definieren zu können. [4]

3.4 Klassifizierungsschema

In Abschnitt 2.1 wurde als Klassifizierungsschema die Formatvorlage der Gang-of-Four für die OO Design Patterns als Beispiel vorgestellt. An diesem Schema haben sich viele Arbeiten über Design Patterns dezentraler autonomer Systeme orientiert und sie an den spezifischen Kontext angepasst. Um den Überblick zu bewahren und die Design Patterns in Abschnitt 3.5 vergleichen zu können wurde folgendes Schema verwendet:

Pattern Name - welchen Namen trägt das Design Pattern?

Aliases - welche alternativen Bezeichnungen existieren für das Design Pattern?

Problem/Applicability - für welche Problemsituation(en) ist das Entwurfsmuster anwendbar?

Solution/Effect - welches Designproblem wird durch das Pattern gelöst bzw. welche Wirkungsweise besitzt es?

Granularity - handelt es sich um ein elementares Entwurfsmuster oder ist es aus mehreren grundlegenden Komponenten zusammengesetzt?

Known Uses - für welches Anwendungsgebiet ist das Pattern bestimmt?

Categorisation - welcher Kategorie gehört das Entwurfsmuster an?

3.5 Überblick über aktuelle Design Patterns

In den letzten Jahren haben verschiedene unabhängige Institute einen Bedarf an Entwurfsmuster für dezentrale autonome Systeme erkannt und begonnen für ihre präferierte Entwicklungsmethodologie und bevorzugten Metamodelle Design Pattern Kataloge [9, 26, 16] zusammen zu stellen. Die Untersuchung ergab dabei, dass die meisten Kataloge für proprietäre Multi-Agenten-Systeme geschaffen wurden, sich die enthaltenen Konzepte aber auch auf dezentrale autonome Systeme übertragen lassen.

Im Rahmen dieser Untersuchung wurden Design Patterns für *Koordinationsmechanismen, Informationsaustausch, soziales Verhalten* sowie Patterns auf *Organisationsebene* ausgemacht. Viele dieser Patterns gehen auf bekannte Problemstellungen der Informations- und Kommunikationstechnologie zurück, greifen aber auch moderne Konzepte selbstorganisierender Systeme auf, wie beispielsweise Stigmergie und Emergenz.

Aus der Menge an untersuchten Entwurfsmustern wird in folgenden beiden Abschnitten eine Auswahl vorgestellt, die entsprechend ihrer Granularität [4] eingeteilt wurden.

3.6 Elementare Patterns

Bei den *elementaren Patterns* handelt es sich um grundlegende Entwurfsmuster, die sich nicht aus anderen Design Patterns zusammensetzen, sondern aus denen andere, komplexeren Entwurfsmustern aufgebaut sind. Die Wahl fiel auf Replication, Evaporation und Aggregation.

3.6.1 Replication

Aliases
Duplication, Redundancy

Problem/Applicability
(1) Wie kann die Zugriffszeit auf Information verringert werden?
(2) Wie kann dem Verlust von Information vorgebeugt werden, im Falle eines Komponentenausfalls verursacht durch böswillige Dritte oder höhere Gewalt?

Solution/Effect
Die Agenten übertragen Information an alle benachbarten Agenten. Ein zusätzlicher Zeitstempel sorgt dafür, dass die Information bei allen Agenten stets aktuell gehalten wird. Über die Umgebung wird Änderungen an einer Information oder neue Information erkannt. Liegt veränderte oder neue Information vor leitet der betroffene Agent die Information an benachbarte Agenten weiter.

Granularity
Es handelt sich um ein elementares Pattern.

Known Uses
(1) Replication kann eingesetzt werden, um threadsichere Bibliotheken zu entwickeln. Die Objekte werden dupliziert und für den Fall eines gleichzeitigen Zugriffs mehrerer unabhängiger Threads auf ein Objekt, wird lediglich die Kopie des Objektes verändert und die Änderung in einem späteren Takt in das Originalobjekt übernommen. [2]
(2) Bei Grafikkarten wird Replication für Doulbe-Buffering eingesetzt, um über einen zweiten Bildspeicher das Flackern des Bildschirms beim Neuzeichnen zu verhindern.

Categorisation
Informationsaustausch.

[4] "Granularität (Körnung), in der Informatik ein Merkmal für die Anzahl von Untergliederungen eines Elements. Die Unterteilung liegt im Bereich von grob bis fein." [7]

3.6.2 Evaporation

Aliases

–

Problem/Applicability

Wie kann über die Zeit Information reduziert bzw. entfernt werden?

Solution/Effect

Über Zeitstempel/Countdown wird veraltete Information von der Umgebung erkannt und über die Zeit entfernt. Letztendlich bleibt keine Information mehr übrig.

Granularity

Es handelt sich um ein elementares Pattern.

Known Uses

Kommt bei stigmergie-basierten Anwendungen, wie beispielsweise Digital Pheromone Paths, zum Einsatz.

Categorisation

Informationsverarbeitung.

3.6.3 Aggregation

Aliases

–

Problem/Applicability

Wie kann die Menge an verarbeiteter Information zusammengefasst werden?

Solution/Effect

Agenten übergeben Informationen an ihre Umgebung. Die Umgebung fasst diese neue Information mit der bestehenden Information zusammen, bevor sie von den Agenten weiter verarbeitet werden.

Granularity

Es handelt sich um ein elementares Pattern.

Known Uses

(1) In Wireless Sensor Networks (WSN) werden Informationen an Knotenpunkten aggregiert, um beim Informationsaustausch den Kommunikationsoverhead zu reduzieren.

(2) Beim Ranking von Produkten im Internet wird nicht jede einzelne Bewertung gespeichert, sondern mit jeder neuen Bewertung der Gesamtwert neu berechnet.

(3) Bei Ameisen wird eine Pheromonspur durch Aggregation jedes Mal verstärkt, wenn eine Ameise die Spur erneut passiert.

Categorisation

Informationsverarbeitung.

3.7 Koordinations- und Kommunikations-Patterns

Neben den elementaren Patterns existiert eine Reihe von *Koordinations- und Kommunikations-Patterns*, die auf den elementaren Entwurfsmustern beruhen. In dieser Arbeit werden *Gradient Fields, Digital Pheromone Paths* und *Market-based Coordination* aus [26] vorgestellt, die im Fallbeispiel des 4. Kapitels zum Einsatz kommen.

3.7.1 Gradient Fields

Aliases

–

Problem/Applicability

(1) Wie können mehrere autonome Einheiten in einem großangelegtem verteilten System räumlich koordiniert werden?
(2) Wie kann Wegewahl (Routing) für Nachrichten und Agenten durchgeführt werden?

Solution/Effect

Über die Umgebung werden räumliche und kontextbezogene Informationen in Form von Feldern (siehe Abbildung 1) zur Verfügung gestellt. Agenten können sich anhand dieser Felder über die Distanz zum Feldmittelpunkt räumlich ausrichten und koordinieren.

Granularity

Aus elementaren Pattern zusammengesetzt.

Known Uses

(1) Gradient Fields können zur räumlichen Anordnung und Bewegung von Agenten verwendet werden.
(2) Kann zur Kontrolle von Charakteren in Videospielen zum Einsatz kommen.
(3) Unterstützt Software-Agenten das World Wide Web zu erforschen.

Categorisation

Koordinations- und Kommunikationsmechanismus.

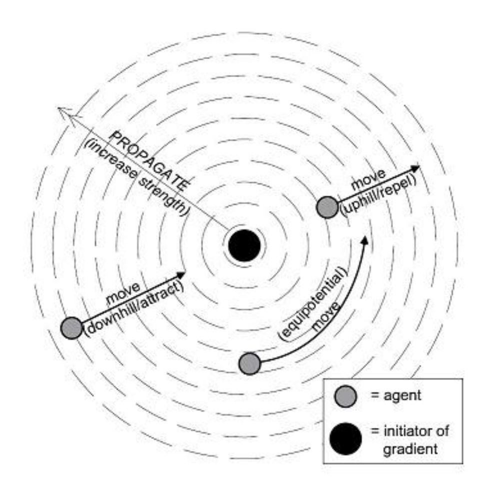


Abbildung 1. Ein Gradient Field mit Ausbreitungsrichtung und Agentenbewegung.

3.7.2 Digital Pheromone Paths

Aliases
Pheromone Trails, Pheromone Field.

Problem/Applicability
Wie kann eine autonome Einheit oder eine Nachricht einen optimalen Weg zwischen Quelle und Ziel zurücklegen?

Solution/Effect
Autonome Einheiten suchen proaktiv nach Zielen, Aufgaben oder Gegenständen und hinterlassen auf ihren Pfaden Pheromonspuren (Stigmergie), an denen sich andere autonome Einheiten orientieren können. Folgen weitere autonome Einheiten diesem Pfad, wird die Pheromonspur verstärkt (Aggregation Pattern) und andere autonome Einheiten auf diesen Pfad gelockt. Über die Zeit verdunsten die Pheromone (Evaporation Pattern) und Pfade verlieren sich, falls keine neue Pheromonspur gelegt wurde.

Granularity
Aus elementaren Pattern zusammengesetzt.

Known Uses
Lösung von graphentheoretischen Problemen wie beispielsweise das Traveling Salesman Problem [18, 10].

Categorisation
Koordinations- und Kommunikationsmechanismus.

3.7.3 Market-based Coordination

Aliases

–

Problem/Applicability

(1) Wie können Ressourcen effizient über dezentrale und verteilte Mechanismen bereitgestellt werden?
(2) Wie werden lokal Informationen über das globale Angebot und Nachfrage von Ressourcen verfügbar?

Solution/Effect

In Anlehnung an die ökonomische Markttheorie wird auf einem virtuellen Markt um Ressourcen gehandelt. Der Preis für die Ressourcen entsteht dynamisch aus Angebot und Nachfrage durch die autonomen Einheiten. Jedes autonome System kann anhand des aktuellen Marktpreises darüber entscheiden, ob es eine Ressource erwirbt oder nicht.

Granularity

Aus elementaren Pattern zusammengesetzt.

Known Uses

(1) Bei der Energieverteilung herrscht Angebot und Nachfrage nach Energie.
(2) Bei der Wegewahl in Netzwerken (Routing) wird um die verfügbare Netzwerkbandbreite gehandelt.

Categorisation

Koordinations- und Kommunikationsmechanismus.

4 Anwendungsbeispiel: Paketlieferdienst

Die Anwendung der vorgestellten Koordinationspattern Gradient Fields, Digital Pheromone Paths und Market-based Coordination werden in der folgenden Fallstudie eines Paketlieferdienstes [26, 24] illustriert. Im Abschnitt 4.1 wird auf die Problemstellung des Beispiels eingegangen und im darauffolgenden Abschnitt 4.2 die Anwendung der Patterns auf den Sachverhalt beschrieben.

4.1 Problemstellung

Kunden können zu beliebiger Zeit einen Paktdienst beauftragen, ein Paket persönlich abzuholen und diese Sendung zu einem gewünschten Empfänger zu befördern. Zum Transport steht eine Reihe selbstständig agierender Lastwagen zur Verfügung, welche sich optimal an die Kundennachfrage anpassen müssen. Auf diese Weise entsteht auf abstrakter Ebene ein dynamisches Netzwerkmodell auf Basis der vom Kunden in Auftrag gegebenen Absender- und Empfangsadressen sowie den Lastwagen, die selbstständig durch die Straßen navigieren müssen (siehe Abbildung 2). Die Herausforderungen sind dabei:

der Versand - jeder neue Kundenauftrag muss einem Lastwagen zugewiesen werden, der für die Abwicklung der Beförderung verantwortlich ist.
die Navigation - jeder Lastwagen muss selbstständig durch die Straßen navigieren können, um die neuen Absenderadressen effizient anzufahren und die Pakete in kürzester Zeit an die richtige Empfangsadresse zuzustellen.

In beiden Fällen treten dabei ständig dynamischen Änderungen auf. Während Kunden jederzeit neue Lieferungen in Auftrag geben können, sollen die Empfangsadressen, bereits in Auftrag gegebener Sendungen, nachträglich geändert werden können. Die Lastwagenfahrer sind zudem den üblichen Straßenverkehrsproblemen ausgesetzt. So können sich Staus bilden oder Hindernisse entstehen, die überwunden werden müssen oder Lastwagen aufgrund einer Panne zum Erliegen kommen. Darüber hinaus sind eine Reihe von Zeitvorgaben einzuhalten:

Abholzeit - bis wann muss das Paket von der Absenderadresse abgeholt worden sein?
Lieferzeit - bis wann muss das Paket zugestellt worden sein?
Pausen - wie lange dauern die Fahrer-Pausen und wann finden sie statt?
etc.

Für dieses hochdynamische Problem, bei dem der Informationsaustausch und die Koordination dezentral zwischen den Lastwagen, Kunden und den Straßen erfolgen, bieten sich autonome dezentrale Systeme als vielversprechende Lösung an.

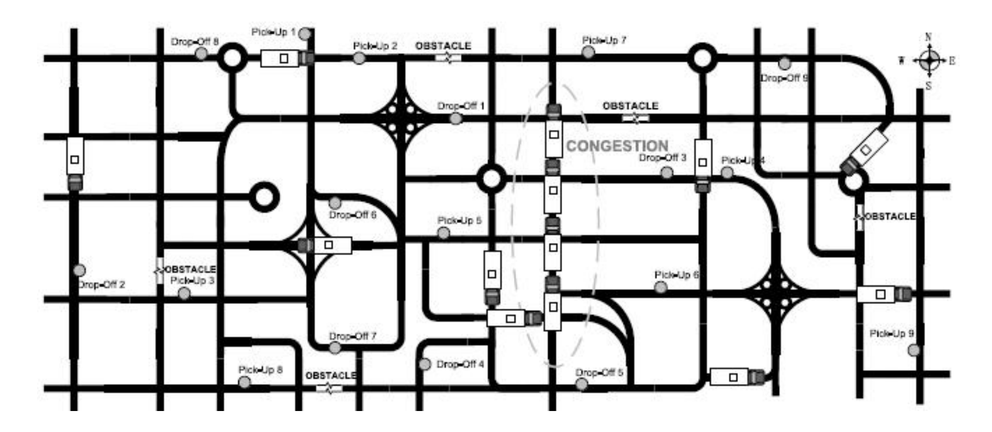


Abbildung 2. Das Paketlieferdienst-Problem nach [26].

4.2 Anwendung der dezentralen Koordinations-Patterns

Der Lösungsansatz für dieses komplexe Problem ergibt sich aus dem Zusammenspiel der einzelnen dezentralen autonomen Teilsysteme innerhalb des vorgestellten Systems. Die Teilsysteme sollen gegenseitig alle notwendigen Informationen

austauschen, um zur Zielerreichung alle notwendigen Entscheidungen und Maßnahmen treffen zu können. Ein Softwarearchitekt muss folglich herausfinden, welche Informationen die autonomen Teilsysteme benötigen, um den Versand und die Navigation effizient durchführen zu können und anschließend entscheiden, welches Koordinations-Pattern für diese Anforderungen in Frage kommt.

Im Falle des Versands muss für jeden eingehenden neuen Auftrag der zu diesem Zeitpunkt *geeignetste Lastwagen* ausgewählt werden. Um aus der Menge an Lastwagen den *geeignetsten Lastwagen* auszuwählen, müssen diese über messbare und vergleichbare Eigenschaften verfügen:

Distanz - wie weit ist der Lastwagen von der Abholadresse entfernt?
Platz - hat der Lastwagen ausreichend Platz für das Paket?
Lieferzeit - wie lange dauert die Zeit für die Zustellung des Pakets?

Besonderes Augenmerk muss dabei auf den verfügbaren Platz der Lastwagen gelegt werden, eine begrenzte Ressource, die den Aufträgen zugeteilt wird. Anhand der Liste an vorgestellten Patterns aus Abschnitt 3.7 fällt die Entscheidung für das *Market-based Coordination Pattern*, welches ein geeignetes Entwurfsmuster für diese Problemstellung sein könnte.

Allerdings müssen für dieses Pattern geeignete Anbieter und Nachfrager identifiziert werden. Im Fallbeispiel [26] werden dazu die Lastwagen als Anbieter von verfügbarem Platz und die Auftrags-Systeme [5] als die Nachfrager des verfügbaren Platz betrachtet.

Als Preismechanismus kommt dabei neben dem verfügbaren Platz die Distanz zur Abholadresse und Lieferzeit in Betracht. Beispielsweise steigt mit der Länge der Lieferzeit oder mit der Knappheit des verfügbaren Platzes gleichzeitig der Preis. Die Auftrags-Systeme können bis maximal zu ihrem Preislimit für den verfügbaren Platz Angebote abgeben und erhalten je nach Marktpreis den Zuschlag. Der Markmechanismus sorgt dafür, dass bei der Zuteilung immer ein globales Optimum erreicht wird.

Im Falle der Navigation wird nach einer Lösung gesucht, welche den Lastwagen ermöglicht sich auf den Straßen zu orientieren. Geeignete Kandidaten für räumliche Koordination sind die *Digital Pheromone Paths* oder das *Gradient Fields Pattern*. Um eine Auswahl treffen zu können, müssen beide Patterns im Detail untersucht und verglichen werden.

Digital Pheromone Paths erweisen sich für die Problemstellung als weniger geeignet, da die Lastwagen dann aktiv nach den Abholadressen, Lieferadressen und den Angeboten suchen müssten. Stattdessen können sich die Lastwagen auf den Straßen mithilfe der Gradient Fields orientieren. Insgesamt kommen drei verschiedene Arten von Gradient Fields zum Einsatz:

Pickup Location Gradients - jede Abholadresse emittiert einen Gradienten, um auf die Anwesenheit eines neuen Auftrags aufmerksam zu machen. Erhält ein Lastwagen den Auftrag, muss er lediglich dem Gradienten folgen.

[5] Die Auftrags-Systeme sind autonome dezentrale Systeme, welche die eingehenden Aufträge repräsentieren.

Delivery Location Gradients - jede Liederadresse emittiert einen Gradienten, sobald ein Lastwagen einen neuen Auftrag erhält. Um die Liederadresse zu erreichen, folgt der Lastwagen diesem Gradienten.

Market Communication Gradients - ermöglichen die direkte Verständigung und Verhandlung zwischen den Lastwagen und den Aufträgen. Dabei werden die Gebote auf dem Markt mittels dieser speziellen Gradienten übertragen, indem sie den Gradienten des Handelspartners folgen. Voraussetzung für die Interaktion ist, dass jeder Markteilnehmer einen *Market Communication Gradient* emittiert.

Mithilfe der vorgestellten *Gradient Fields* lassen sich ebenfalls die Sonderfälle wie die Behandlung von Staus und Hindernisse abdecken. Einen Spezialfall stellt dagegen eine Lastwagen-Panne dar. In diesem Fall müssten beispielsweise die umliegenden Lastwagen mittels zusätzlicher *Gradient Fields* über den Unfall informiert werden und die enthaltenen Pakete zusammen mit dem Auftrag je nach verfügbaren Platz auf die anderen Fahrzeuge aufgeteilt werden. Zudem stellt die Unfallstelle ein zusätzliches Straßenhindernis dar, an dem Staus entstehen können.

Das Anwendungsbeispiel zeigt, dass mit einem umfangreichen Pattern-Katalog die Designprobleme eines dezentralen autonomen Systems reduziert werden können. Ein Softwarearchitekt kann über die Design Patterns auf das Wissen anderer Experten zurückgreifen und sich auf die wesentlichen Aufgaben konzentrieren, um sein Ziel zu erreichen.

5 Zusammenfassung und Ausblick

Zu Beginn dieser Arbeit wurde ein Überblick über objektorientierte Design Patterns der Gang-of-Four und der GRASP Patterns gegeben und eine klassische Formatvorlage zur Klassifizierung von Patterns vorgestellt. Nach der Einführung und Erläuterung der Eigenschaften *autonom*, *dezentral* und *selbstorganisierend*, wurden auf das Agentenorientierte Softwareengineering (AOSE), sowie deren Entwicklungsmethodologien eingegangen.

In Abschnitt 3.3 wurde klargestellt, dass spezielle Design Patterns für autonome dezentrale Systeme notwendig sind und diese sich, aufgrund der unterschiedlichen Abstraktionsebenen, nicht zwingend mit den objektorientierten Patterns überschneiden.

Anhand des Fallbeispiels eines Paketlieferdienstes wurden abschließend gezeigt, dass während die Komplexität bei der Softwareentwicklung weiter zunimmt, autonome dezentrale Systeme die Möglichkeit bieten diese Komplexität besser zu beherrschen. Vielversprechende Ansätze liefert dazu der Forschungsbereich *Organic Computing* [20, 15].

Literatur

[1] C. Alexander, S. Ishikawa, and M. Silverstein. A pattern language - towns-building-construction, 1977.

[2] J. Andrews. Threading the ogre3d render system, 2006. http://cache-www.intel.com/cd/00/00/33/13/331357_331357.pdf.
[3] B. Bauer. Vorlesung model-driven software development. Universität Augsburg, Sommersemester 2006.
[4] C. Bernon, M. Cossentino, M.-P. Gleizes, P. Turci, and F. Zambonelli. A study of some multi-agent meta-models. In J. Odell, P. Giorgini, and J. P. Müller, editors, *AOSE*, volume 3382 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2004.
[5] Bibliographisches Institut & F. A. Brockhaus AG Brockhaus Naturwissenschaft und Technik, Mannheim und Spektrum Akademischer Verlag GmbH, 2003.
[6] G. Booch, J. Rumbaugh, and I. Jacobson. *Das UML-Benutzerhandbuch*. Addison-Wesley, 1999.
[7] Der Brockhaus: in 15 Bänden, Permanent aktualisierte Online-Auflage. Leipzig, Mannheim: F.A. Brockhaus 2002-2007.
[8] W. Cunningham and K. Beck. Using pattern languages for object oriented programs. Technical report, OOPSLA - Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1987.
[9] T. de Wolf and T. Holvoet. A catalogue of decentralised coordination mechansims for designing self-organising emergent applications, August 2006.
[10] M. Dorigo and L. Gambardella. Ant colonies for the traveling salesman problem. *In BioSystems*, 1997.
[11] J. Weiß. Duden der Informatik. Ein Fachlexikon für Studium und Praxis, 2003.
[12] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, 2003.
[13] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, published as Intelligent Agents III*, volume 1193, pages 21–35. Springer-Verlag, 1996.
[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
[15] H. Kasinger. Ein mda-basierter ansatz zur entwicklung von organic computing systemen. Technical report, Universität Augsburg, 2005.
[16] M. Kolp, P. Giorgini, and J. Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In *ATAL '01: Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, pages 128–140, London, UK, 2002. Springer-Verlag.
[17] C. Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, October 2004.
[18] E. Lawler, J. Lenstra, A. RinnooyKan, and D. Shmoys. The travelling salesman problem, 1985.
[19] C. Müller-Schloer. Informatiklexikon der gesellschaft für informatik e.v., 2004. https://www.gi-ev.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/58/.
[20] C. Müller-Schloer, H. Schmeck, and T. Ungerer. Antrag auf einrichtung eines neuen dfg-schwerpunktprogramms, 2004. Organic Computing.
[21] W. Pree. Design patterns for object-oriented software development, 1995.
[22] W. Reif and H. Seebach. Vorlesung softwaretechnik. Universität Augsburg, Wintersemester 2005/2006.
[23] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., 1991.

[24] M. W. P. Savelsbergh and M. Sol. The general pickup and delivery problem. *Transportation Science*, 1995.

[25] G. Weiss. Agent orientation in software engineering, 2003.

[26] T. D. Wolf and T. Holvoet. Design patterns for decentralised coordination in self-organising emergent systems. In S. Brueckner, S. Hassas, M. Jelasity, and D. Yamins, editors, *ESOA*, volume 4335 of *Lecture Notes in Computer Science*. Springer, 2006.

[27] M. Wooldridge and N. Jennings. Agent theories, architectures, and languages: A survey. In *ECAI Workshop on Agent Theories, Architectures, and Languages*, volume 890, pages 1–39, 1994.

[28] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

Emergentes Fehlverhalten

Andriy Ostroverkhov

Universität Augsburg
andriy.ostroverkhov@uni-augsburg.de

Zusammenfassung Schnelle Entwicklung von Computer und Softwaresystemen kann sehr schlecht mit der alten Design- und Enwicklungsmethoden von diesen Systemen zusammenleben. Die neuen Ansätze wie Selbst-Organisation und Emergenz werden von den Forschern immer öfter verwendet. Die neuen Systeme werden immer mehr autonomer und selbständiger, was aber sher oft zu unvorhersagbarem Verhalten führt. Dieses Verhalten kann nicht nur unvorhersagbar sondern auch unerwünscht sein.

1 Einleitung

Die Entwicklung von Software- und Computersystemen, und damit verbundene steigende Komplexität von diesen Systemen benötigt die anderen Ansätze für Entwicklung, Design, Evaluierung, Integration und Kontrolle. Früher konnte man leicht das Systemverhalten vom Verhalten der einzelnen Teilen von diesen Systemen vorhersagen. Somit könnte man auch eine zentralisierte Steuerung für diese Systeme benutzten. Jetzt werden solche Systeme mehr komplexer und frühere Ansätze sind nicht mehr ausreichend. Adaptive Algorithmen, Machine Learning und künstliche Intelligenz sind einige der Ansätzen, die als Basis für die neuen Designmethoden von solchen Software- und Computersystemen diente.

Schon heute weisen viele moderne verteilte Systeme steigend verflochtene Struktur[13] (z.B. ad-hoc Netzwerke, Transportsysteme usw.). Unterschiedliche Subsysteme hängen von vielen anderen Subsystemen, wirken aufeinander durch die komplexen, dynamischen und unvorhersagbaren Wege. Auch mehr und mehr von Systemen brauchen dezentralisierte Kontrolle und Struktur (z.B. ad-hoc Netzwerke) und können ihre Anforderungen nur autonom erfüllen.

Zentralisierte Steuerung, die heute viele existierende Systeme besitzen, hat viele Nachteile wie Komplexitätsengpässe und sehr niedrige Robustheit. Komplexitätsengpass führt nicht nur zum Stress und Verwirrung, er begrenzt sehr stark die Progressgeschwindigkeit. So ist es offensichtlich, dass die neuen komplexen Systeme sollen dezentralisiert gesteuert werden.

Wir bräuchten aber dafür einen radikal anderen Ansatz, um diesen Engpass zu überwinden. Ein Schritt nach vorne ist *Autonomic Computing Initiative* von IBM. Als Inspiration haben die IBM Forscher autonomes Nervensystem von den Menschen genommen. Ihre Vorschläge dafür, wie man das für die neuen Systeme nutzen kann, basiseren auf Feedback-, Adaptation- und Steuerungsmodelle, die

das erste Mal im Jahre 1950 vorgeschlagen wurden. Dieser "kybernetische Ansatzt"ist sehr interresant, für die zukunftigen Systeme braucht man etwas sogar mehr radikales, wie Selbstorganisation.

Der Begriff *Selbstorganisation* wird später in diesem Artikel erläutert, jetzt können wir sagen, dass die Systeme, die Selbstorganisation aufweisen, haben so genannte Selbst-X Eigenschaften, und zwar Selbst-Konfiguration, Selbst-Heilung, Selbst-Optimierung und Selbst-Schutz. Zukunftige Systeme sind somit mehr unabhängig, flexibel und selbstständig. Die Systeme mit solchen Selbst-X Eigenschaften werden oft als Selbst-Managing Systeme genannt. Entwicklung und Design von solchen Systemen ist aber nicht eine einfache Aufgabe.

Es werden sehr oft die Beispiele aus der Natur entlehnt, um die Systeme mit Selbst-X Eigenschaften zu bauen. Solche biologischen Systeme, wie Ameisen, Bienen, Termiten, Fisch- und Vögelschwärwe sind einige Beispiele, wo man die oben erwähnten Selbst-X Eigenschaften sehr schnell identifiziert. In solchen Systemen gibt es immer mehrere einfache Objekte, die grundsätzlich nur mit einander kommunizieren. Es gibt kein zentrales Objekt, das für Systemsteuerung und Kontrolle verantwortlich ist. In solchen biologischen Systemen kann man aber auch eine andere Eigenschaft bemerken. Das Verhalten von einzelnen Objekten lässt sich nicht das Verhalten von dem System im Ganzen zu verstehen. Der globale Zustand von solchen Systemen kann man nicht von den Zuständen der Objekten vorhersagen. Dieses Phänomen ist unter dem Begriff *Emergenz* bekannt.

Die Systeme, die Emergenz aufweisen, haben somit emergentes Verhalten. Wobei die Natursysteme haben ihr Verhalten seit vielen Jahrhunderts verbessert und angepasst, die von Menschen produzierenden künstlichen Systeme haben sehr oft das Verhalten, das weit vom Ideal ist. Emergentes Verhalten von künstlichen Systemen kann sehr oft zum unerwünschten Konsequenzen führen und wird als emergentes Fehlverhalten genannt. Was emergentes Fehlverhalten ist, wie kann man damit umgehen und welche Klassifikationsmethode dafür existieren, wird in diesem Artikel untersucht und beschrieben.

Der Rest von diesem Artikel ist auf folgende Weise gebaut: im Abschnitt 2 werden die Begriffe Selbst-Organisation und Emergenz erklärt und vergliechen. Danach werden die Begriffe emergentes Verhalten und emergentes Fehlverhalten dargestellt. Im Abschnitt 3 stellen wir einige Beispiele von emergentem Fehlverhalten aus unterschiedlichen Bereichen dar. Danach, im Abschnitt 4 gehen wir auf die Klassifikation von emergentem Fehlverhalten und desses Ursachen ein. Weiter werden einige Techniken für Umgang mit emergentem Fehlverhalten genannt. Am Schluß kommt die Zusammenfassung und Zukunftsausblick.

2 Definitionen

Bevor wir auf die Definition von emergentem Fehlverhalten eingehen, definieren wir die Begriffe wie Selbst-Organisation, Emergenz und auch emergentes Verhalten.

2.1 Selbst-Organisation

Die intuitive Definition von Selbst-Organisation, die im Jahr 1998 von Dempster [4] gegeben wurde, lautet wie folgt:

> *Self-organisation refers to exactly what is suggested: systems that appear to organise themselves without external direction, manipulation, or control.*

Die Organisation bezieht sich vor allem auf die Ordnung im Systemverhalten.

Selbst-Organisation bedeutet vor allem die spontane dynamisch produzierte (Re-)Organisation. Weiter sind einige Definitionen genannt, die sich auf unterschiedliche Verhaltensweise von selbst-organisierenden Systemen beziehen.

Bonabeau [1] btrachtet die Schwärme, als ein Beispiel für selbst-organisirende Systeme. Für Schwärmverhalten definiert er folgende Mechanismen, die das identifizieren:

- Vielfachwechselwirkung zwischen Individuen;
- Rückwirkendes positives Feedback (Steigerung von Pheromonen, wenn Nahrung gefunden ist);
- Rückwirkendes negatives Feedback (Verdämpfung von Pheromonen);
- Steigerung von Verhaltensmodifikation (Steigerung von Pheromonen, wenn die neuen Pfade gefunden sind).

Prigogine und seine Kollegen [11] haben vier notwendigen Anforderungen für Systeme mit selbst-organisierendem Verhalten unter externem Druck identifiziert.

- *Mutual Causality*: Mindestens zwei Systemkomponenten haben die Kreisbeziehungen, eine beinflußt die andere.
- *Autocatalysis*: Mindestens eine der Komponenten ist kausal von den anderen beeinflusst, was auch zu ihrem Wachstum führt.
- *Far-from Equilibrium Condition*: Das System importiert eine große Energiemenge von außen und verwendet diese Energie, um seine Struktur zu erneuern, und eher zerstreut als speichert die zuwachsende Unordnung (Entropie) zurück in die Umgebung.
- *Morphogenetic Changes*: Midenstens eine der Systemkomponenten muss für die zufälligen Außenvariationen ausgesetzt sein.

Die anderen Forschungen für Selbst-Organisation in Multi-Agenst-Systemen haben zwei Definitionen von selbst-organisierenden Systemen definiert[22]:

- Starke selbst-organisierende Systeme sind die Systeme, die ihre Organisation ohne irgendeiner expliziten externen bzw. internen zentralisierten Steuerung bauen;
- Schwache selbst-organisierende Systeme sind die Systeme, in den die Reorganisation unter der zentralisierten internen Steuerung oder Planung passiert.

Darüberhinaus schließt Selbst-Organisation auch Organisation ein. Diese Organisation beinhaltet die Strukturänderungen vom System und Verhaltensänderungen von Komponenten. Das selbst-organisierende System demgemaß nicht nur steuert oder adaptiert sein Verhalten, das erzeugt und ständig ändert seine Struktur bzw. Organisation.

Die selbst-organisierenden Systeme sind in der Wirklichkeit sehr robust: sie können die ganze Reihe von Fehlern, Störungen oder sogar Zerstörungen widerstehen. Wenn Systemschaden zu groß sind, verschlechtern sich die Funktionen vom System, allerdings ohne plötzliche Zerstörung bzw. Systemabbruch.

Stabilität und Widerstandsfähigkeit von den selbst-organisierenden Systemen bedeutet allerdings nicht, dass sie statisch oder rigid sind. Solche Systeme adaptieren ihre Organisation zu jeweiligen Umgebungsänderungen, lernen neue Tricks, um mit nicht vorgesehenen Problemen zu kooperieren. Ändert sich die Umgebung, adaptieren die mit der Umgebung zusammenwirkenden Systemkomponenten sofort ihre Zustände, um das System wieder in Ordnung zu bringen. So wird das System ständig reorganisiert.

Je mehr Störungen passieren, desto größer die Anzahl von moglichen Konfigurationen, die das System untersucht, und desto besser wird der Endzustand vom System. Der Kybernetiker von Foerster [24] hat dieser Prinzip als *order from noise* und der Thermodynamiker Prigogine [11] hat das als *order through fluctuations* genannt.

Naturbeispiele von selbst-organisierenden Systemen sind allbekannte soziale Insektensysteme, wie Ameisen, Termiten und Bienen. Die Kommunikation in diesen Systemen entsteht mittels Pheromonen, die in der Umgebung ausgebreitet werden. Die anderen Beispiele von selbst-organisierenden Tieren sind Vögel- oder Fischschwärme.

Die künstlichen selbst-organisierenden Systeme sind sehr oft von den Naturbeispielen inspiriert. Dabei die Schwärme sind die meist verbreiteten Quellen für diese Inspiration.

2.2 Emergenz

Typisch wird Emergenz als ein Phänomen beschrieben, bei dem das globale Verhalten von den gegenseitigen Einwirkungen der lokalen Teilen des Systems entsteht. Um die Idee von Emergenz zu illustrieren, präsentieren wir zuerst einige Beispiele von Systemen, wo dieses Phänomen zu finden ist.

Das erste Beispiel stammt aus dem Naturbereich und bezieht sich auf die Ameisen. Diese Ameisen haben die Rolle, in der sie die Umgebung untersuchen und Futter suchen. Wenn sie Futter finden, gehen sie zurück zum Ameisenhaufen und markieren den Weg mit Pheromonen. Der kürzeste Pfad ist die Struktur, die aus der gemeinsamen Aktivität von Ameisen emergiert. Dieser Pfad ist die Realität nur für den Beobachter des Systems, die Ameisen sehen den nicht.

Die Erscheinung von Gewissen ist ein Beispiel der Emergenz für Menschen. Das Gewissen wird vom Searle [21] als eine Eigenschaft des Gehirnes auf dem höheren Niveau betrachtet. Biologisch gesehen ist das Gehirn ein komplexes System, das aus Neuronen und der Kommunikation zwischen ihr besteht. Diese

Neuronen sind das niedrigste Niveau. Zur Zeit können wir das Gewissen nicht verstehen oder erklären, wenn wir nur die Neuronen und ihrer Zusammenspiel betrachten.

Das Ergebnis von Emergenz wird oft als Phänomen genannt, und kann eine Struktur bzw. Framework wie Bénard Zelle, Verhalten wie Segelflugzeug im Conways Spiel des Lebens, oder eine Funktion (Aufgabe) wie Bildung vom Kurszeitplan von einigen lokalen Elementen sein.

Folgende Eigenschaften muss das System erfüllen wenn es Emergenz aufweist:

- Es muss zwei Niveaus (Mikro- und Makroniveau) im System geben;
- Das System muss mindestens auf dem Makroniveau wahrnehmbar sein;
- Keine Reduzierbarkeit von Eigenschaften des höhen Niveaus zu den Eigenschaften des niedrigen Niveaus;
- Gegenseitige Abhängigkeiten zwischen den Niveaus:
  - Makroniveau zwingt das Mikroniveau;
  - Mikroniveau verursacht das Makroniveau;
- Das System muss Neuheit aufweisen: etwas neues, was früher nicht existierte, wird produziert;
- Es muss auch Zusammenhang aufweisen, in der Sinne, dass es sein eigenes Identität hat;
- Es wird auch mit den Teilen stark verknüpft, die Emergenz produzieren.

Hat das System die Objekte mit linearen Aktivitäten, wird die Erklärung und die Vorhersagemöglichkeit des globalen Zusatandes möglich. Um umergentes Phänomen zu bekommen, braucht man im Gegenteil nicht lineare Aktivitäten auf dem Mikroniveau.

2.3 Vergleich von Selbst-Organisation und Emergenz

Emergenz und Selbt-Organisation sind zwei Phänomene, die sowohl Ähnlichkeiten, als auch Unterschiede haben. Diese zwei Begriffe werden sehr oft falsch gegenseitig benutzt.

2.3.1 Ähnlichkeiten Da die Emergenz und die Selbst-Organisation sehr unterschiedliche Aspekte von Systemverhalten hervorheben, gibt es zwischen ihnen nur wenige Ähnlichkeiten. Die Hauptähnlichkeit ist, dass Emergenz und Selbst-Organisation beide dynamische Prozesse sind. Die beide sind auch robust. Jedoch die Robustheit der Emergenz bezieht sich auf die Flexibilität in bestimmten Teilen, die emergente Eigenschaften verursachen. So ein Ausfall des einzigen Teiles führt nicht zum Systembruch. Die Robustheit der Selbst-Organisation bezieht sich auf die Anpaßbarkeit und die Möglichkeit den gesteigerten Zustand zu unterstützen. Zu wenig Ähnlichkeiten schließt allerdings nicht aus, dass die beiden Konzepte verwandt sind. Sie ergänzen einander wenn sie kombiniert werden.

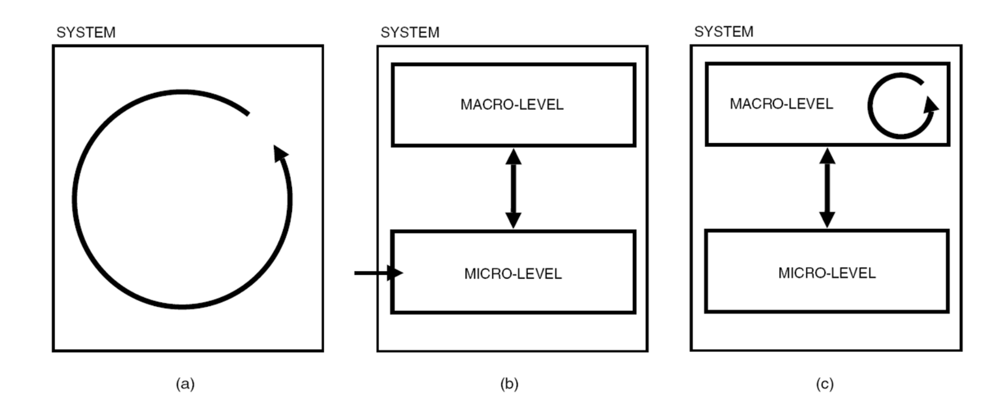


Abbildung 1. (a) Selbst-Organisation ohne Emergenz; (b) Emergenz ohne Selbst-Organisation; (c) Emergenz mit Selbst-Organisation. *Quelle: De Wolf, Holvoet* [3]

2.3.2 Unterschiede Oben wurde es gezeigt, dass Emergenz und Selbst-Organisation unterschiedliche Eigenschaften von einem System betonen. Die beiden Konzepte können in Isolation existieren. Erstmal betrachten wir Selbst-Organisation ohne Emergenz, danach Emergenz ohne Selbst-Organisation. Die Abbildung 1(a) illustriert schematisch ein System mit Selbst-Organisation, aber ohne Mikro-Makro-Effekt. Es gibt keine externe Kontrolle vom System. Die gekrümmten Pfeilen repräsentieren das interne Organisationsprozess. Die Eigenschaften, die für Emergenz spezifisch aber für Selbst-Organisation nicht nötig sind, sind radikale Neuheit, Mikro-Makro-Effekt, Flexibilität mit Beziehung auf Elementen und dezentralisierte Steuerung. Falls eine der Eigenschaften fehlt, ist es nicht der Fall der Emergenz.

Abbildung 1(b) illustriert schematisch eine andere Situation. Das System hat Mikro-Macro-Effekt, aber es ist nicht selbst-organisierend. Die wesentlichen Eigenschaften sind hier *increase in order*, keine Außenkontrolle und Anpassungsfähigkeit.

Emergenz ohne Selbstorganisation ist bestimmt möglich. Zum Beispiel, in Physik kann Thermodynamik aus der statistischen Mechanik in einem stationären (und somit nicht selbstorganisierenden) System emergieren [23]. Nehmen wir Gas als ein Beispiel von Material, das ein bestimmtes Volumen im Raum hat. Dieses Volumen ist eine emergente Eigenschaft, das von den Wechselwirkungen (z.B. Attraktion und Repulsion) zwischen den individuellen Partikeln resultiert. Jedoch solch ein Gas ist in einem stationären Zustand. Die statistische Komplexität bleibt derselbe im Laufe der Zeit, d. h. die Partikeln können ihre Positionen ändern, aber der Betrag der Struktur bleibt derselbe. In diesem Fall haben wir ein System, dessen anfängliche Bedingungen genug sind, um emergente Eigenschaften aufzuweisen.

Anpassungsfähigkeit bezieht sich auf das Bedürfnis, ein Gleichgewicht zwischen der Auswahl an einem spezifischen Verhalten und der Rücksicht einer großen Vielfalt von Verhalten zu erreichen. Foukia und Hassas [9] formulieren das im Bezug auf ein Gleichgewicht zwischen der Erforschung und Ausnutzung. Ein System kann Chaos aufweisen (z.B. das Betrachten einer großen Vielfalt von Verhalten und auch ständige Umschaltung zwischen ihnen) das aus den Wechselwirkungen zwischen den Mikroniveauteilen emergierte. So ein System ist aber nicht selbstorganisierend, weil es sich nicht organisiert, um eine Sonderfunktion zu fördern.

2.4 Emergentes Verhalten und emergentes Fehlverhalten

Das Begriff *emergentes Verhalten* ist sehr leicht vom Begriff *Emergenz* nachzuvollziehen. Genau wie Emergenz kann "emergentes Fehlverhalten durch die Analyze von den Niveaus, die einfacher als das System im ganzen sind, nicht vorhersagen werden".

Wenn die einzelnen Teile vom System ein anderes Verhalten als das System im ganzen haben, und das Verhalten vom System ist nicht die Summe von den Verhalten der Teilen, so spricht man über emergentes Verhalten. Das Konzept "emergentes Fehlverhalten" im Gegenteil ist komplexer und kann sowohl zu breit als auch zu eng verwendet werden. Um zu verstehen, was man als emergentes Fehlverhalten nennen soll, brauchen wir einige Tests durchzuführen.

Wenn wir aus der Definition von emergentem Verhalten oben ausgehen, können wir sehr leicht bestimmte Fälle von Fehlverhalten definieren, die sicherlich nicht emergent sind:

- *Bugs von Einzelnkomponenten*, die das ganze System abbrechen: wenn eine kritische Komponente des Systems einfach nicht mehr funktioniert, wird es einen Systemabbruch erwartet.
- *Inhärent uneffiziente Algorithmen*: manche algorithmische Aswahlen sind voraussagbar uneffizient. Zum Beispiel, das replizierte Dateisystem, das mit den Repliken eher seriell als parallel kommuniziert, wird wohl sub-optimale Perfomanz aufweisen. Man kann sie vorhersagen, ohne das zu wissen, wie die Repliken verhalten.
- *Mangelhafte Resourcen*: Primitive Resourcen (z.B. CPU, Speicher, Netzwerklatenz und Bandbreite, Speicherkapazität, Latenz, und Bandbreite) inhärent verhindern das System von der Perfomanz auf dem gefordeten Niveau. Zum Beispiel, man kann nicht 1 Gigabyte Data über 56 Kbit/sec Dialup in 1 Minute senden.

Obwohl Dyson [5] schreibt, dass "emergentes Verhalten, nach der Definition, ist etwas, was geblieben ist, nach dem schon alles erklärt wurde", es scheint nicht genug zu sein, emergentes Fehlverhalten so zu definieren, wie etwas, was nicht zum vorhersagbaren Verhalten passt. Hier ist eine Liste mit der einigen Eigenschaften von emergentem Fehlverhalten:

1. *Inhärent schwer vorhersagbares Verhalten*: wenn sogar die Regeln eines Systems, die sein Verhalten bestimmen, sind bekannt und deterministisch, es kann sehr schwer sein, das ganze Verhalten vom System vorherzusagen; wenn das System auch die wahrscheinlickeitbasierten oder nicht-linearen Komponente beinhaltet, sehr großen Zustandsraum hat bzw. seine Skalierbarkeit ganz groß ist, das Problem der Vorhersage wird sogar größer.
2. *Plötzliche Verhalaltensänderungen*: wenn das Verhalten vom System sich schnell zwischen Modis mit sehr unterschiedlichen Perfomanzeigenschaften ändern kann, ist sein Verhalten sehr schwer vorherzusagen, wenn die Parametern, die diesen Modus steuern, beim kritischen Punkt sind. Zum Beispiel, *Ethernet Capture Effect* (3.3.1) steigert "plötzlich" wenn Chipdesigners das Inter-Packet Gap zum bei Spezifikation erlaubten Minimum verringern.
3. *Verstärkung von anscheinend unwichtigem Verhalten*: Vorhersage ist eifacher, wenn wir unwichtige Abweichungen von erwateten Verhalten ignorieren können. Wenn diese kleine Abweichungen durch die solchen Efekkte wie Resonanzen oder Koinzidenzen verstärkt werden können, können sie zum unvorhersagbaren Verhalten führen.

Eine schwierige Frage ist, ob chaotisches Fehlverhalten besser zu verstehen als emergegentes Fehlverhalten ist oder doch nicht. John Holland [14] schreibt die Definition von Emergenz wie folgt:

> *I'll not call a phenomenon emergent unless it is recognizable and recurring: when this is the case, I'll say the phenomenon is regular. That a phenomenon is regular does not mean that it is easy to recognize or explain*

Diese Definition beinhaltet kein chaotisches Verhalten, da es um die periodischen Erreignise geht. Auch Parunak und VanderBok [18] unterscheiden explizit "emergentes Chaos"von der echten Zufälligkeit. Nicht alles emergentes Verhalten ist chaotisch. Zum Beispiel, in Router Synchronisierung (Abschnitt 3.3.2) gibt es ein Problem, dass konvergentes Verhalten ohne Rücksicht auf Ausgangsbedinungen aufgekommen ist.

3 Beispiele von emergentem Fehlverhalten

In diesem Abschnitt stellen wir einige Beispiele von emergentem Fehlverhalten in unterschiedlichen Bereichen dar.

3.1 Non-Computerbereich

Am ersten Tag der Eröffnung von der Millennium Footbridge haben die unerwarteten übermaßigen Quervibrationen zum Erschweren vom Fußgängerbewegungen geführt. Die Konstrukteure haben nicht erwartet, dass die Synchronisierung von einzelnen Schritten mit einander und mit der eigenen Bewegung der Brücke zum diesen Effekt kommt.

Dieses Problem ist uns wirklich interessant, da die Brückenkonstrukteure diese Art von Problemen ziemlich gut kennen. Man sollte auch den verrufenen Abbruch von der Tacoma Narrows Bridge seit vier Monaten nach der Eröffnung erwähnen. Dieser Fall ist jedem Brückenkonstrukteur in der Welt auch sehr bekannt. Das Brückenform hat in Wind genügenden Auftrieb generiert, um die Hauptschwingungen zu induzieren. So sieht man, dass sogar in den technischen Bereichen mit mehrjährigen Erfahrungen man auch emergentes Fehlverhalten treffen kann.

Ein anderes Beispiel von emergentem Verhalten ist Autoverkehr: das wird als Ergebnis von der Entscheidungen der Fahrer, Fußgänger und Verkehrskontrolleuren produziert. Die Verkehrsstaus wird meistens als Ergebnis von emergentem Fehlverhalten betrachtet.

3.2 Computerhardwarebereich

Wir sind daran gewöhnt, dass die Festplatten mit einander nur durch die Storage Controllers bzw. Storage Protokolle (wie z.B. SCSI) kommunizieren. In den größeren Installationen sind mehrere Festplatten auf den Racks montiert. Auf solche Weise können die Leistungen von einer Festplatte durch die Vibrationen von Nachbarnfestplatten beeinflusst werden. Die Hersteller berücksichtigen es bei der Produktion von solchen Festplatten, auch deswegen kosten sie mehr als die normale Festplatten.

3.3 Computernetzwerkeberiech

Computernetzwerkbereich bietet viele Beispiele von emergentem Fehlverhalten, wahrscheinlich deswegen, weil die Netzwerke sehr oft groß sind und ihre Leistungsfähigkeitsprobleme weit verbreitete Effekte aufweisen. Jedoch einige Beispiele unten passieren bei der kleinsten Netzwerkgröße: Zwei-Knoten Netzwerken.

3.3.1 Ethernet Capture Effect Der *Ethernet Capture Effect* ist ein gutes Beispiel von emergentem Fehlverhalten. Capture Effect erzeugt signifikante Unklarheit in bestimmten CDMA/CD Umgebungen. Betrachten wir den Fall von einem kurzen LAN mit genau zwei Host-Rechner A und B, beide sind mit vielen Paketen zum Senden. A und B werden gleichzeitig das Kanal als frei erkennen, beiden werden Pakete senden und die Kollision verursacht die Berechnung von einem zufälligen Backoff. Nehmen wir an, dass A einen kleineren Backoffwert als B auswählt. Danach wird A ein Paket senden und die beiden Hosts sehen das Kanal wieder als frei. Die beiden senden wieder gleichzeitig Pakete, was eine neue Kollision verursacht. Jedoch der Host A hat die letzte Kollision gewonnen, sein Kollisionszähler wurde zurückgesetzt und das erwartete zufällige Backoffwert von B ist größer als das von A. Insofern wird A wahrscheinlich wieder gewinnen und die Chancen von B werden immer verschlechtern.

3.3.2 Router Synchronisierung Die Routers tauschen periodisch ihre Routing Protokollnachrichten. Man glaubt, dass in einem großen Netzwerk ein konstantes Hintergrundniveau von Routing Protokollnachritenverkehr gibt. Floyd und Jacobson [8] haben aber gezeigt, dass in einem Netzwerk, wo es scheinbar unabhängige periodische Prozesse gibt, können diese Prozesse versehentlich synchronisiert werden. Den Übergang passiert aber nicht stufenweise, sondern plötzlich. So ist es schwer voraus zu sehen, wenn jemand nicht sorgfältig nachsieht. Sie haben auch gezeigt, dass die Synchronisierung auch mit der Einführung von einer genügend großen zufälligen Komponente in der Routing Protokoll Updating Zeit vermieden werden kann.

3.3.3 Route flap damping in BGP Das Border Gateway Protocol (BGP) schafft den Basis für Routing zwischen ISP im Internet. Internetlinks gehen jede Zeit herauf und hinab, und generieren damit potenziell viel BGP-Updateverkehr, und diese Instabilität kann auch die Bearbeitungskapazitäten von Internetrouter überlasten. BGP beinhaltet soweit *Route Flap Damping* (RFD) Mechanismus, um die Verbreitung von Routinginformation zu begrenzen.

Mao hat gezeigt, dass der standarte RFD-Mechanismus "bedeutsam die Konvergenzzeiten von relativ stabilen Routern verschlimmern kann" [16]. Resultierende Konvergenzzeit kann bis zu 60 Minuten sogar nach einem Routerabzug sein.

Das Problem, dass Mao Stämme von einer Wechselwirkung zwischen zwei BGP Mechanismen identifiziert: der Routerabzugprozess und der Mechanismus, der Stabilität der ganzen Infrastruktur sichert. Sie zeigen, dass der mehr allgemeine Fall dieses Problems (*withdrawal-triggered suppression* genannt) in Topologien unter einer bestimmten kritischen Größe nicht erscheint. D.h. dieses Problem ist eine Folge des globalen Netzwerkarchitektur (oder mindestens eines gemäßigt großen Subgraphen) und seinen Wechselwirkungen mit lokalen Implementierungsauswahlen.

Die von Mao durchgefürte Analyse von einfachen Topologien ermöglichte sie, die Frequenz der withdrawal-triggered suppression im echten Internet zu schätzen. Obwohl die existierenden BGP-Traces nicht direkt die Fälle dieses Problems zeigen, Mao hat verstanden, dass sie die wahrscheinlichen Fälle der *withdrawal-triggered suppression* von einer charakteristischen Unterschrift schließen konnten. Sie haben herausgefunden, dass das relativ oft vorkommen kann.

3.3.4 TCP's Nagle Algorithmus Zwei Hosts-Rechner, die die Daten über TCP tauschen, können das schlechte Zusammenspiel zwischen dem TCP Nagle Algorithmus von der Absenderseite und dem Delayed-Acknowledgment Algorithmus von der Empfängerseite erfahren. Diese Wechselwirkung reizt vom traditionellen Design vom Netzwerkstack. Das Problem ist aber nicht nur rein akademisch; die Benutzer stoßen sehr oft darauf, besonders wenn sie über die Netzwerke kommunizieren, deren maximale Paketengröße größer als Ethernetgröße ist.

3.4 Bereich der verteilten und Betriebssystemen

3.4.1 Falschkonfigurirter Load Balancer Abbildung 2 zeigt uns die Stuktur von einer einfachen Multi-Tiered verteilten Apllikation mit vielen über Internet verteilten Klienten, Front-End Server, einem Load-Balancer, zwei Applikationservern und zwei Datenbankservern. Die ganze Apllikation sammelt periodische Messreports von den Klienten, macht die Bearbeitung auf den Applikationservern und danach speichert die Reports in dem replizierten Dantenbank.

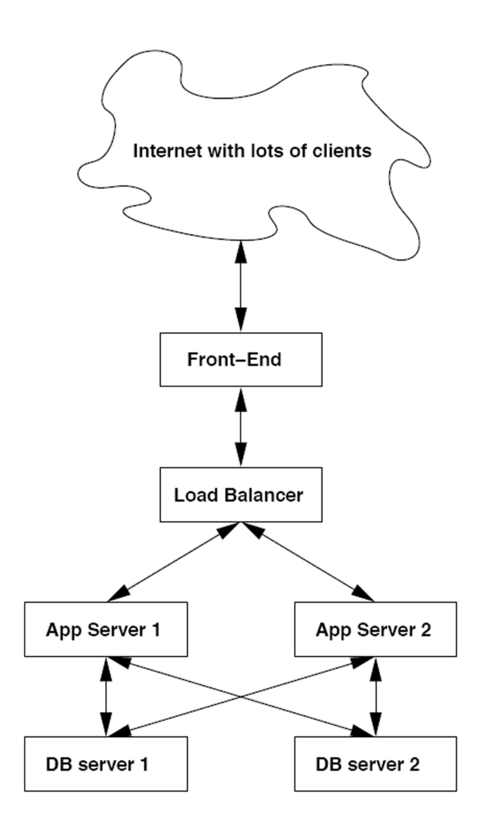


Abbildung 2. Beispiel von multi-tiered verteiltem System *Quelle: Mogul* [17]

Der Load-Balancer in dem System hat zwei Aufgaben: er verteilt gleichmäßig den Arbeitslast zwischen verfügbaren Applikationservern, erkennt die Fehler des Applikationservers und unterbricht die Aufgabensendung. Der Load Balancer erkennt den Fehler, wenn der Applikationsserver auf Anfrage während bestimmtes Timeouts nicht antwortet. Vermuten wir, dass das System perfekt funktioniert, wenn das das erste Mal gestartet wird. Nach einigen Monaten erhöhen Datenbanklatenzzeiten und das Effizienzindex verschlechtert sich.

Gehen wir davon aus, dass der Load Balancer wurde so kofiguriert, relativ kleine Timeouts für Erkennung des Applikationserverausfalles zu benutzen. An einer Stelle haltet das System auf, die Anfragen zu beantworten. Datenbanklatenzen erhöhen sich zu dem Punkt, wenn die Applikationserver antworten nicht mehr den Load Balancer während des konfigurierten Timeouts, und Load Balancer erkennt die beiden Server als außer Betrieb. Als die Variante von diesem Problem, der Load Balancer wechselt zwischen Applikationservern ab; als jeder Server ist gezwungen volles Laden zu bearbeiten, seine Latenzzeit überschreitet das Timeout, wobei der andere entladene Server scheint repariert zu sein. Offensichtlich, dass das System im ganzen sich falsch verhaltet. Jedoch keine von Komponenten hat einen Feheler aufgewiesen. Man kann sagen, dass jemand ein falsches Wert für Load Balancer ausgewählt hat, dieser Fehler ist aber nicht so klar wenn das System zum ersten Mal getestet wird.

4 Klassifikationen

4.1 Klassifikation von emergentem Fehlverhalten

Hier werden allgemeine Kategorien von emergentem Fehlverhalten in Softwaresystemen betrachtet. Die Liste kann folgendes enthalten:

- ***Flattern***: Wettbewerb über die verdichteten Resourcen die aber im Raum zerstreut sind, wobei die Kosten für Umschalten zwischen verdichteten Gruppen dominiert über der möglichen nutzlichen Arbeit. Es ist auch nützlich das Form von Flattern zu unterscheiden, das durch die bessere Koordination bzw. Plannung vermieden werden kann von dem Form, bei dem klar ist, dass System mehr Resourcen braucht, um die Aufgaben zu erfüllen.
- ***Unerwünschte Synchronisierung***: Hier geht es um eine Reihe von Systemen, deren zeitlich wechselndes Verhalten unkorreliert werden soll, wobei sie versuchen, sich immer zu korrelieren. D.h. die auf dem statischen Multiplexing basierte Resourcenzuweisung kann durchfallen. Das Millenium Footbridge Problem, von Floyd und Jacobson beschriebene Synchronisierung der Routingnachrichten alle kommen in diese Kategorie.
- ***Unerwünschte Oszillation oder Periodizität***: Das System pendelt zwischen den Zuständen, wegen dem zufälligen bzw. schlecht entwickelten Feedback-Loop zwischen mehreren Komponenten.
- *Deadlock*: Der Progress bleibt wegen den Kreisabhängigkeiten stehen. Deadlock kann sicherlich in einem System auftreten, wo jede Komponente richtig funktioniert.
- ***Livelock***: Der Progress bleibt stehen wegen zwei oder mehreren Threads, die trotzt ständigen Änderungen keinen Netzprogress erreichen können. Emfanglivelock ist ein Sonderfall, in dem der Durchsatz vom System abnimmt (sogar bis Null), wobei die Eingangsrate steigert ab einem bestimmten Punkt. Ein Empfangslivelock unterscheidet sich von einem Deadlock insofern, dass der Durchsatz wiederherstellt wird, wenn die Eingangsrate abnimmt.

– ***Phasenubergang***: Das Systemverhalten ändert sich radikal während einige Variablen ändern sich stufenweise. In den anderen Bereichen, wie Physik, solche plötzlichen Änderungen können durch einen Phasenübergang modelliert werden. Einige Computersysteme können auch einen Phasenübergang aufweisen. Zum Beispiel, drahtlose Ad-Hoc Netzwerke haben sehr oft kritische Schwellen.

Diese Klassifikation enthält keine *Fehler* oder *Komponentenausfälle*. Tatsächlich, kein in diesem Artikel beschriebenes Beispiel von emergentem Fehlverhalten stammt von einem Komponentenausfall. Die Ursachen liegen bei der Entwicklung bzw. der Implementierung von einem System.

4.2 Ursachen emergentes Fehlverhaltens

Wenn wir den Fall von emergentem Fehlverhalten aus der Liste oben erkennen können, ist es nur ein Schritt zur Problemlösung. Wir brauchen auch die Klassifikation von Ursachen des emergenten Fehlverhaltens, die einem bestimmten Fall entsprechen. Zum Beispiel, die Ursache von Flattern könnte einfaches Memory-Leak sein. Das könnte auch ein Überprüfungsfehler von der Zulassungskontrolle sein, was zur Zulassung von vielen Jobs in System mit begrenzten Resourcen führt. Das könnte auch der Implementierungsbug in einem Zeiplannungsalgorithmus sein, der versucht, die schlechten Zeitplannungsfälle zu vermeiden, macht aber das Gegenteil. Für jede Kategorie aus dem Abschnitt 4.1 sollte es möglich sein, die Liste von generischen Ursachen zu bauen. Diese Liste könnte man für die Ursache des emergenten Fehlverhaltens benutzen. Die Liste ist wahrscheinlich nicht erschöpfend: Systeme können auch einzigartiges Fehlverhalten aufweisen – es sollte aber doch möglich sein, bestimmte Anzahl von Fehler zu bedecken.

Emergentes Fehlverhalten ist der Aspekt von ganzem System und nicht nur von einer Koponente, deswegen viele von der Ursachen aus der Liste unten können viele Komponente involvieren. Hier ist die mögliche Liste von den Ursachen des emergenten Fehlverhaltens:

– ***Unerwartete Resourcen Sharing***: Der Designer hat vermutet, dass die einzelnen Komponente den Zugriff nur auf separate Resourcen hatten, wobei sie aber Resourcen gemeinsam nutzen und diese Resourcen sind nicht ausreichend. Wir denken über die Resourcen als über die Informatikabstraktionen wie Berechnungen, Speicher und Kommunikationmedia. Echtweltsysteme haben auch die Probleme wegen den unerwarteten gemeinsamen Resourcennutzung, wie Energieversorgung oder Abkühlungssysteme.
– ***Massive Scale***: Einige Kommunikationskomponene im System sind groß genug, um dem globalen Verhalten Aufstieg zu geben, wenn sogar einzelne Komponente ein einfaches Verhalten haben.
– ***Dezentralisierte Kontrolle***: Allgemein schätzen wir ein dezentralisiertes System mehr als ein zenralisiertes System, obwohl wir wissen, dass die Zentralisierung es oft einfacher macht, das System zu implementieren und zu

steuern. Huberman und Hogg [15] haben eine theoretische Analyse durchgeführt, wie ein verteiltes System ohne zentralisierte Steuerung von unvolständigem Wissen und verzögerter Information leiden kann, und somit Oszillieren und Chaos aufweisen.

- ***Miskonfiguration***: Viele Fehler in komplexen Systemen, wie inkonsistente Netzwerk Routing Policies oder Backupsysteme, kann man zur Miskonfiguration zählen. Während man behaupten kann, dass die Miskonfiguration einfach das Form des Implementierungsfehlers, in Wirklichkeit ist das häufig einfach schwierig für Bediener, die globalen Konsequenzen ihrer lokalen Konfigurationswahlen zu verstehen. Zentralisiertes Konfigurationmanagement kann manchmal dieses Problem vermeiden, aber kann nicht für Systeme anwendbar sein, die aus vielen administrativen Domänen zusammengesetzt sind.
- ***Unerwartete Eingaben oder Ladungen***: Viele Systeme reagieren sehr schlecht auf unerwartete Eingaben oder unerwartete Ladungen. Nicht alle solche Fehlverhaltenarten sind emergent. Denken Sie jedoch daran, dass es schwierig sein kann, die Grenzen "des Systems als Ganzes"zu definieren, und manchmal Entwickler beschreiben sie zu detailliert im Vergleich zu dem, wofür sie verantwortlich sind.

Sowohl Parunak und VanderBok als auch Huberman und Hogg weisen darauf hin, dass die Verzögerung einer der Hauptbeitragender von emergentem Fehlverhalten ist. Verzögerung ist inhärent in verteilten und Netzwerksystemen. Wobei es scheinen kann, dass die primäre unerwünschte Folge von Latenz die langsame Ausführung vom System ist, könnte Latenz sogar mehr schädlich sein, weil es Verständnis und Kontrolle von einem System verschlechtert.

Wie Huberman und Hogg hinweisen, Verzögerung bedeutet, dass kein einzelner Gesichtswinkel völlig konsequenten und aktuellen Ansicht vom globalen Systemzustand haben kann; das ist was zu Schwingungen und Chaos führt.

5 Umgang mit emergentem Fehlverfahren

5.1 Erkennung von emergentem Fehlverhalten

Hat man die Klassifikationen von emergentem Fehlverhalten und seinen Ursachen, so kann man auch die Techniken für Erkennung und auch für Ursachendiagnose entwickeln. In vielen Fällen ist das das Beste, was wir tun können, wenn man davon ausgeht, dass emergentes Verhalten ursprünglich unvorhersagbar ist.

Um die Erkennung zu unterstützen, kann das Betriebssystem oder die Infrastruktur von verteilten Systemen ihre Applikationen für allgemeine Übereinstimmung mit solchen Mustern, wie Flattern, Livelock, unerwünschte Periodizität usw. beobachten. Dieser Ansatz hat Erfolg mit solchen Techniken gezeigt, wie eine von Romer für dynamischen Seitenmapping [20] beschriebene Technik. Parunak und VanderBok beschreiben mehrere Mehrzwecktechniken, um emergentes Verhalten in Steuerungssystemen zu erkennen, die auf ihren Gliederung von Ursachen basiert [18]. Zum Beispiel, periodisches Verhalten kann durch die

Fourieranalyse entdeckt werden; ähnliche Techniken könnten von Betriebssystemen und ihren assoziierten Managementsystemen eingesetzt werden.

Das Diagnoseproblem ist schwerer zu lösen. Ein Ansatz wäre, die Systemdesignererwartungen zum Diagnosesystem auszustellen. Patrick Reynolds mit seinen Kollegen hat ein System entwickelt (mit dem Namen Pip), um Verhaltensprobleme in verteilten Systemen zu diagnostizieren [19]. Im Pip-Ansatzt drückt der Programmierer seine Erwartungen über die Systemleistung und kausale Struktur aus, inklusiv sowohl lokale als auch pfadbasierte globale Erwartungen. Eine Middlewareschicht kontrolliert dann Anwendungsverhalten (einschließlich die Kommunikation zwischen Knoten), um verletzte Erwartungen zu entdecken. Dieser Ansatzt baut sich auf Perl und auf der Überprüfungstechnik von Perfomanzfeststellung für parallele Anwendungen.

Wichtig ist, dass der Pip-Ansatz von der Fähigkeit der Programmierer die formellen (und richtigen) Spezifizierungen zu schreiben, nicht abhängt. Pip versucht nicht, den empirischen Ansatzt zu vermeiden, nur es weniger schmerzhaft zu machen, und Programmierer leicht zu zwingen, die Möglichkeit unerwartetes Verhalten zu konfrontieren.

Systementwickler konnen hier damit helfen, mehr Überwachungen und Logging ins System einzubauen. Somit könnten die Diagnosetools einen globalen Sicht von Systemverhalten zu bauen, und so könnte das unerwünschte Verhalten erfasst werden. Systementwickler neigen Hinzufügung von solchen Überwachungen wegen den Run-Timekosten zu vermeiden, aber die Kosten von einem Systemausfall können viel größer und sicherlich weniger vorhersagbar sein. (Das Space Shuttle Programm stellt zur Verfügung ein klares Beispiel: der Shuttle wurde seit zwei Dekaden eingesetzt bevor NASA dafür entschied, Kameras am Bord zu verwenden, um zu sehen, dass Schaumstoff aufgelöst wurde).

5.2 Kontrolle des emergenten Fehlverhaltens

In manchen Situationen ist es unmöglich, emergentes Fehlverhalten zu diagnostizieren bzw. eine Diagnose kann darauf zeigen, dass die Ursache nicht direkt behoben werden kann.

An diesen Stellen sind Techniken für Verbesserung bzw. Umgang mit emergentem Fehlverhalten nötig. Zum Beispiel, Floyd und Jacobson [7] zeigen, wie die Einführung von zusätzlicher Zufälligkeit ins Routing-Update Timing die unerwünschte Synchronisierung abbbrechen kann. Sie haben sogar "berechnet" wieviel Zufälligkeit braucht man dafür. Die Einführung von Zufälligkeit ist eine übliche Methode: Parunak und VanderBok beschreiben [18], wie diese Zufälligkeit im Timing die Probleme mit fehlerhaften Schweißnahten löst, die mit automatisierten Punktschweissenroboter produziert sind.

Ebenso, obwohl ist es theoretisch möglich das Netzwerkstack zu modifizieren, um Livelocks zu vermeiden, in der Praxis kann es keinen Zugriff auf Quellkode geben. In dieser Situation kann Livelock doch durch die Einführung von einem Rate-Limiting Box stromaufwärts zum Livelock verhindert werden [17].

Gribble [12] schlägt einige Designstrategien vor inkl. die Benutzung von systematischen Overprovisioning, Einlaßkontrolle, Selbstprüfung und geschloßene

Kontrolleschleifen für die Adaptation. (Jedoch die Praxis, wie von Parunak und VanderBok mitgeteilt wurde, schlägt vor, dass die Einführung von Kontrollschleifen kann eventuell das Problem mit emergentem Fehlverhalten nicht lösen.) Gribble schlägt auch vor, eher solche Systeme zu entwickeln, die Fehler schon im voraus erwarten und können sie schnell beheben, als die fehlerlosen Systeme zu bauen.

George Candea[2] weist hin, dass die ganze Systemabhängigkeit durch die unvorhersagbar verhaltenden Komponente verringert werden kann. Er schlägt zwei Möglichkeiten vor, wie man das erreichen kann: entweder durch die Begrenzung von der Verbreitung des unvorhersagbaren Verhaltens, oder durch den Schutz der Komponenten von der unerwarteten Eingabe. Zum Beispiel, Softwaresicherungen (wie Brandmauer) werfen die rahmenübergreifenden Eingaben ab, bevor sie verletzbare Komponente erreichen; Ausgabewächter erkennen scheinbaren Komponentenausfall und brechen verdächtige Module ab, und dadurch zwingen Byzantiner zum Fail-Stop Verhalten.

Jedoch nimmt der Vorschlag von Candea an, dass Fehlverhalten entweder bei der Komponenteneingabe oder bei der Komponentenausgabe offenbar ist; systemweites (emergentes) Fehlverhalten könnte auf diesem Niveau entweder unsichtbar sein, oder könnte so durchdringend sein, dass Softwaresicherungen oder Wächter das ganze System effektiv zumachen würden. Ein Schutzt gegen emergentem Fehlverhalten ist eher mehr als eine Form von Dämpfung (um die Verbreitung des Problems zu verringern) oder Festklemmen (um die Menge des Schadens zu beschränken) anzunehmen.

Das Ziel von vielen Forschungen von verteilten Systemen war die Entwicklung von komplexen Systemen, die immer funktionieren: sowohl durch die grundlegenden Designprinzipien (z.B. *two-phase commit and replication*), als auch durch die bessere Engineering (z.B. Modelüberprüfung und Type-Safe Sprachen).

5.3 Vorhersage und Vermeidung von emergentem Fehlverhalten

Es ist sehr oft mehr wichtig, eine vorhersehbare als eine optimale Leistungsfahigkeit zu haben. Wenn die Leistungsfahigkeit vorhersehbar aber gleichzeitig suboptimal ist, kann man bereit sein, die vorausgesehene Uneffizienz zu bezahlen. Wenn aber die Leistungsfahigkeit manchmal unvorhersehbar schlecht ist, der Systembesitzer kann in der Situation geraten, das Worst Case zu erleben.

Das Vorhersage von der Leistungsfahigkeit bedeckt viele Bereiche. In vielen Situationen kann es sehr nutzlich sein, die Moglichkeit zu haben, emergentes Fehlverhalten zu vorhersagen. Das Konzept *Vorhersage von emergentem Fehlverhalten* selber kann oxymoronisch klingeln, da einige Definitionen von emergentem Verhalten sind von Anfang an unvorhersehbar sind. Dieses scheinbares Paradox hat mindestens zwei möglichen Lösungen. Die Definition von emergentem Fehlverhalten, die es als "*unvorhersehbares durch die Analyse auf allen Niveaus, die einfacher als das System allgemein sind*" beschreibt, hat die Vorhersagemöglichkeit von den Verfahren, die auf dem globalen Niveau gelten. Zweitens, wobei es unmöglich wird das spezifische emergente Fehlverhalten vorher zu sagen, es kann doch möglich vorhersagbar sein, dass System auf bestimmtes nicht spezifiziertes

emergentes Fehlverhalten anfällig ist. Drittens, kann es moglich sein, emergentes Fehlverhalten anhand von globalen Symptomen vorher zu sagen.

5.4 Testen auf emergentes Fehlverhalten

Es ist überhaupt nicht wichtig, wie gut wir die Techniken für Vermeidung, Diagnose, Verbesserung oder Reparieren von emergentem Fehlverhalten entwickeln können, die Komplexität von jeweiligen Situationen kann den Aufwand leicht verwirren. Es kann scheinen, dass Problem mit emergentem Fehlverhalten gelöst wurde, wobei das nur ein wenig verbirgt wurde.

Deshalb brauchen wir auch die Techniken, um das System auf emergentes Fehlverhalten testen zu können. Das Testen von einem komplexen System hat immer große Herausforderungen. Zum Beispiel, Armando Fox [10] schlägt vor, dass die Bedienungen, die zu emergentem Fehlverhalten führen, sind nicht immer während des Testens bekannt oder erkennbar. Die Lösung kann solche Techniken zum Reproduzieren von letztem emergentem Fehlverhalten, oder eher der Konfiguration beinhalten, die zu dem geführt hat. Es wäre auch möglich die Bedienungen automatisiert, basierend auf der Klassifikation von Ursachen (Abschnitt 4.2) zu generieren. Die anderen Herausforderungen beinhalten die Notwendigkeit der automatisierten Entdeckung des emergenten Fehlverhalten (Abschnitt 5.1), weil die extensiven Testingprotokolle automatisiert werden sein müssen, und können nicht auf Menschen verlassen, um festzustellen, ob ein Test durchgefallen ist.

Selbstverständlich können die domänespezifischen Techniken der meistzweckmäßige Ansatzt fürs Testen sein. Wie im Abschnitt 3.3 demonstriert wurde, die Routingprotokolle, wie BGP können komplexes und unerwünschtes Verhalten aufweisen, das teilweise als emergent klassifiziert werden kann. Nick Feamster und Hari Balakrishnan [6] haben gezeigt, dass die statische Analyse von BGP-Konfigurationsinformation durch den ganzen ISP und nicht durch jeden Router, viele BGP-Kofigurationsfehler erkennen kann. Sie schreiben, dass “diese Fehler waren von sehr einfachen Single-Router-Fehler ... bis zum komplexen, netzwerkbreiten Fehler, der Zusammenspiel zwischen meheren Router involviert”.

6 Zusammenfassung und Ausblick

Wir werden nie können, alle emergenten Fehlverhaltenprobleme zu lösen, besonders mit der ständigen Erhöhung von Komplexität. Wir konnen und sollen aber im Zustande sein, die wiederkehrenden Fehlverhaltenmuster zu erkennen und genug von der vergangenen Erfahrung zu lernen, damit diese Muster zu vermeiden und auszubessern. Da wir brauchen, die globale Sicht zu bauen, spielen die Forschungen von Computersystemen besonders im Erkennen von emergentem Fehlverhalten eine bedeutsame Rolle.

Einige Firmen haben die ehrgeizige Perspektive für die Zukunft von der komplexen Computersystemen geäußert, was sie mit Unmöglichkeit erklärt haben, diese Systeme immer wieder mit der älteren Methoden erfolgreich zu steuern.

Diese Perspektive konfrontiert mit dem emergentem Fehlverhaltenproblem. Das ist kein unüberwindliches sondern ein unvermeidliches Problem.

So, zum Beispiel, hat IBM seine Perspektive von *Autonomic Computing* geäußert, wobei die Systeme selbst-konfigurierend, selbst-optimierend und selbst-heilend sind. HP hat *Adaptive Enterprise Perspektive* geäußert, wo die IT-Umgebung die schnellen Änderungen in Business-Level Strategien und Taktiken unterstützt.

Literatur

[1] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm intelligence: from natural to artificial systems.* Oxford University Press, Inc., New York, NY, USA, 1999.

[2] G. Candea. Predictable software - a shortcut to dependable computing? Technical report, Stanford University, http://arxiv.org/abs/cs.OS/0403013, 11 March 2004.

[3] T. De Wolf and T. Holvoet. Emergence versus self-organisation: different concepts but promising when combined. In S. Brueckner, G. Di Marzo Serugendo, A. Karageorgos, and R. Nagpal, editors, *Engineering Self Organising Systems: Methodologies and Applications*, volume 3464 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, 2005.

[4] M. Dempster. A self-organising systems perspective on planning for sustainability. Master's thesis, University of Waterloo, School of Urban and Regional Planning, 1998.

[5] G. B. Dyson. *Darwin Among the Machines: The Evolution of Global Intelligence.* Perseus Books Group, 1998.

[6] N. Feamster and H. Balakrishnan. Pip: Detecting the unexpected in distributed systems. *In Proc. NSDI*, pages 43–56, May 2005.

[7] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *SIGCOMM Comput. Commun. Rev.*, 23(4):33–44, 1993.

[8] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Trans. Netw.*, 2(2):122–136, 1994.

[9] N. Foukia and S. Hassas. Towards self-organizing computer networks: A complex system perspective, 2003. in: G. Di Marzo-Serugendo, A.Karageorgos, O.F. Rana and F. Zambonellini (Eds), proceeding of AAMAS'2003 Workshop on Engineering Self-Organizing Applications, 15 July 2003, Melbourne, Australia. pp 77-83. To appear.

[10] A. Fox. Personal communication, 2005.

[11] P. Glansdorff and I. Prigogine. *Thermodynamic Theory of Structure Stability and Fluctuations.* Wiley and Sons, London, 1971.

[12] S. D. Gribble. Robustness in complex systems. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 21, Washington, DC, USA, 2001. IEEE Computer Society.

[13] K. Herrmann, G. Mühl, and K. Geihs. Self-management: The solution to complexity or just another problem? *IEEE Distributed Systems Online*, 6(1), 2005.

[14] J. H. Holland. *Emergence From Chaos to Order.* Oxford Univ Press, October 1998.

[15] B. A. Huberman and T. Hogg. The behavior of computational ecologies. In B. A. Huberman, editor, *The Ecology of Computation*, pages 77–115. North-Holland, Amsterdam, 1988.

[16] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route flap damping exacerbates internet routing convergence. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 221–233, New York, NY, USA, 2002. ACM.

[17] J. C. Mogul. *Emergent (Mis)behavior vs. Complex Software Systems.* EuroSys, 2006.

[18] H. Parunak and R. VanderBok. Managing emergent behavior in distributed control systems. In *ISA Tech '97.* Instrument Society of America, 1997.

[19] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[20] T. H. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Operating Systems Design and Implementation*, pages 255–266, 1994.

[21] J. Searle. *The Rediscovery of the Mind.* MIT, Cambrige, MA, 1992.

[22] G. D. M. Serugendo, M.-P. Gleizes, and A. Karageorgos. Agentlink first technical forum group self-organisation in multi-agent systems. *AgentLink Newsletter*, Issue 16:23–24, 2004.

[23] C. R. Shalizi. *Causal architecture, complexity and self-organization in time series and cellular automata.* PhD thesis, THE UNIVERSITY OF WISCONSIN - MADISON, 2001.

[24] H. von Foerster. On self-organizing systems and their environments. In M. Yovitts and S. Cameron, editors, *Self-Organizing Systems*, pages 31–50. Pergamon, New York, 1960.

Review ausgewählter Projekte im SPP1183 Organic Computing Phase I

Michael Buthut

Universität Augsburg
michael.buthut@student.uni-augsburg.de

Zusammenfassung Das von der Deutschen Forschungsgemeinschaft gegründete SPP 1183 läuft bereits seit knapp zwei Jahren. Das Programm umfasst 18 Projekte und ist in zwei Phasen gegliedert. In Phase I werden die projektspezifischen Grundlagen für die Zielerreichung gelegt. In Phase II werden dann die gewonnen Erkenntisse in den jeweiligen Anwendungsszenarien umgesetzt, sofern das Projekt in Phase II fortgeführt wird. Auf neun ausgewählte Projekte wird im Folgenden eingegangen, davon werden Organic Traffic Control und QE in einem detailierten Fokus betrachtet.

1 Einleitung

In der Vergangenheit war die heutige Vielzahl komplexer Geräte schwer vorstellbar, dennoch hat sich diese Diversität der Geräte durchgesetzt. Die Vorzüge der modernen Technik, die immer und überall verfügbar scheinen, werden sehr geschätzt. Aber genau dadurch entstehen neue Herausforderungen. Fortschreitende Vernetzung intelligenter und komplexer Systeme ermöglicht die Entwicklung vieler neuen Anwendungen. Aber gleichzeitig beherbergt die Vernetzung das Problem der Kontrolle und der Beherschbarkeit dieser Systeme. Um die Kontrolle und Steuerung dieser Systeme zu erleichtern sollen diese selbständig und verlässlich im Hintergrund agieren. Aus dieser Vision heraus wurde der Fachbereich Organic Computing geschaffen. Organic Computing ist ein Konzept, dass versucht derartige Systeme robust, sicher, zuverlässig, flexibel und vertrauenswürdig zu machen. Vor allem ist auch eine deutlichere Orientierung an den Bedürfnissen der Anwender dringend notwending. Dazu werden zukünftige Systeme autonomer handeln, d.h. sie werden selbst lebensähnlich aufgebaut sein und lebensähnlich handeln. Deshalb werden derartige Systeme organisch genannt. Organic Computing Systeme sind Systeme die sich den jeweiligen Umgebungsbedürfnissen dynamisch anpassen können und die Eigenschaft besitzen, selbst-organisierend, selbst-konfigurierend, selbst-optimierend, selbstheilend und selbst-schützend zu sein. Inspiration für diesen jungen Fachbereich sind dabei Phänomene und Verhaltensstrukturen aus der Natur. Die Bedeutung und die Aktualität der Organic Computing Ziele wurden zunächst durch die Organic Computing Initiative erkannt, einer gemeinsamen Initiative der Gesellschaft für Informatik (GI) und der Informationstechnischen Gesellschaft (ITG)

des Vereins Deutscher Ingenieure (VDI). Ziel dieser Initiative ist es, die technologischen Entwicklungen der nächsten Jahre abschätzen zu können, und darüber hinaus die für die Informatik entstehenden wissenschaftlichen Herausforderungen bezüglich der Gestaltung zukünftiger Informations- und Kommunikationssysteme abzuleiten. Die von der GI und ITG initiierte Forschungsinitiative hat aus diesem Grund das Schwerpunktprogramm Organic Computing der Deutschen Forschungsgemeinschaft geschaffen, in dem z. Zt. 18 Projekte, darunter zwei aus Augsburg, gefördert werden. Im Rahmen dieser Arbeit werden neun ausgewählte Projekte des SPP 1183 [8] in einem Review abgehandelt.

2 Organic Computing Grundlagen

Organic Computing ist im Moment immer noch ein konzeptionelles Gerüst, da es noch nicht vollständig entwickelt und immer noch ein aktuelles und viel versprechendes Forschungsgebiet ist. Doch schon heute ist absehbar, dass es ein ernst zu nehmendes Teilgebiet der Informatik werden wird. Oberstes Ziel ist das Verständnis organischer Strukturen und die ihr Zugrunde liegenden molekularen, kognitiven und sozialen Verhaltensweisen. Dieses Wissen soll zur maschinellen Nachahmung verwendet werden um komplexe Systeme, die aus verschiedensten, teilweise heterogenen, Teilsystemen bestehen besser beherrschbar zu machen.

Wichtigster Bestandteil des Fachgebietes sind die „Selbst-X“ Eigenschaften.:

- Selbstkonfiguration
- Selbstoptimierung
- Selbstheilung
- Selbstschutz

sowie

- Umgebungsbewusstsein
- vorausschauendes Handeln

Ähnliche Anforderungen bzw. Eigenschaften formulierte Paul M. Horn bereits bei IBM als Konzept des „Autonomic Computing“ [15] [10]. Damit sollten die Architekturen der Server weitaus „selbständiger“ und den durch aus traditionellen Ingenieursdisziplinen entwickelten Computer Systemen überlegen sein. Allerdings liegt der Fokus dieses Konzepts auf Mainframe Architekturen und deckt nur einen kleinen Teil bestehender Systeme ab. Organic Computing geht einen Schritt weiter und verwendet die Autonomic Prinzipien und erweitert diese um die Erforschung natürlicher Phänomene wie z.B. Selbstorganisation und emergentes Verhalten. Die Konzepte sollen grundlegende Design-Kriterien erzeugen mit denen zukünftige Systeme den Anforderungen genügen können.

Die Selbstorganisation stellt dabei eine der wichtigsten grundlegenden Anforderung dar:

> *„A self-organizing system not only regulates or adapts its behavior, it creates its own organization.“ [4]*

Ein Beispiel eines selbstorganisierenden Systems ist ein Ameisenstaat [1]. Einzelne Individuen werden erst durch die Selbstorganisation und dem zusammenarbeitenden Verhalten zu einem komplexen System, das vielfältigste Aufgaben übernehmen kann. Das ganze System bildet in der Betrachtung auf der Makroebene eine andere Darstellung als auf der Mikroebene. Dieser Mehrwert des Gesamtsystems bildet die Emergenz.

Unter Emergenz versteht man:

> *„Emergence is defined as a property of a total system which cannot be derived from the simple summation of properties of its constituent subsystems. Emergent phenomena are characterized by* 1 *the interaction of mostly large numbers of individuals* 2 *without central control with the result of* 3 *a system behaviour, which has not been pprogrammedëxplicitly into the individuals.“ [17]*

Ein Beispiel für eine emergente Eigenschaft ist die Resonanzfrequenz eines Schwingkreises. Die beiden Bestandteile des Schwinkreise, die Spule und der Transistor haben einzeln keine Resonanzfrequenz. Erst ihr Zusammenwirken erzeugt diese Eigenschaft [1]. So hat das gesamte System auf Makroebene andere Eigenschaften als auf der Mikroebene, bei der die Systemteile einzeln betrachtet werden.
In der Natur finden sich viele Vorbilder, deren Verhalten durch das Verständnis genetischer bzw. evolutionärer Algorithmen nachgeahmt werden soll. Man befasst sich daher z.B. mit der Erforschung von „Schwarmintelligenz“ und versucht die gewonnen Erkenntnisse dabei auf verteilte Systeme zu übertragen, um die Koordination und die Kommunikation zu verbessern.

3 Review der Projekte im SPP1183 Organic Computing Phase I (Teil 2)

Im folgenden werden sieben ausgewählte Projekte des von der Deutschen Forschungsgemeinschaft (DFG) über eine Laufzeit von 6 Jahren mit einer Summe von 10,5 Mio € geförderten SPP1183 vorgestellt [20] und die Ergebnisse der Projekte aus Phase I bewertet.

3.1 Smart Teams: Local Distributed Strategies for Self-Organizing Robotic Exploration Teams

Dieses Projekt legt seinen Fokus auf die Selbstorganisation einer Gruppe von mobilen Robotern die ein unbekanntes Terrain erforschen. Die Robotergruppe (Smart Team) soll dabei ohne zentrale Steuerung ihre aufgetragene Aufgabe erfüllen. Jedes Individuum hat dabei nur lokales Wissen und eine beschränkte Kommunikationsreichweite. Denkbare praktisches Einsatzgebiet wären für ein solches Team Rettungs- oder Aufklärungsaufgaben.

3.1.1 Hintergrund und Ziele Das Smart Team sollen sich nur mit einfachen lokalen Regeln selbst organisieren, aber dabei sein Handeln immer dem globalen Ziel unterordnen. Um ein unbekanntes Terrain zu erforschen soll dabei ein robustes und energiesparendes Kommunikationsnetz gespannt werden. Die Teams sollen möglichst lange ihre Aufgabe erfüllen. Um dies gewährleisten zu können muss der Energiebedarf jedes Teil- und des Gesamtsystems minimiert werden. Dabei muss sowohl die Erforschung optimal als auch der Engergieverbrauch für die Bewegung eines Roboters optimal sein. Es müssen dazu nicht nur die theoretische Grundlage und die Algorithmen analysiert und verbessert werden, sondern auch die Qualität der Strategie im praktischen Einsatz ermittelt werden. Exploration und Kommunikation sind die Themen welche in Phase I hautpsächlich umgesetzt werden sollen. Energieeffizienz und die Ressourcenverteilung sollen in Phase II vertieft werden.

3.1.2 Ergebnisse Um nun die fundierte Analyse der Strategien vornehmen zu können, wurden die Problemstellungen formal definiert. Beispielsweise wird die zu erforschende Gegend als Graph repräsentiert, in dem die Knoten die Lokationen und die Kanten die Erreichbarkeit selbiger widerspiegeln. Dabei ist es nicht zwingend notwendig, dass alle Kanten durchlaufen werden müssen. Adjazente Kanten können sich von den anderen Kanten unterscheiden. Der Graph ist aber zu Beginn nicht bekannt und muss erst aufgebaut werden [19]. Jeder Roboter sieht dabei nur adjazente Kanten und kann mit anderen Robotoren kommunizieren, die sich innerhalb des selben Knotens befinden. Die Roboter haben nur drei Möglichkeiten ihre Aktionen in einem Schritt auszuführen, sie können entweder kommunizieren, lokale Berechnungen durchführen oder sich bewegen bzw. verweilen. Die zu optimierenden Paramter sind dabei die maximale Anzahl an Kantenüberquerungen sowie die Gesamtzeit der Exploration. Die Ergebnisse dieser Untersuchung sind, dass die Abbildung für die Gesamtzeit wichtig, aber für die Energieeffizienz nicht in einem vergleichbaren Maß gravierend ist [19]. Für die effiziente Kommunikation wird eine Kommunikationskette aufgebaut in der die Roboter, die sich zwischen den Enden befinden, als Relay fungieren. Oberstes Ziel dabei ist aber die Kommunikation in jeder Situation aufrecht zu erhalten. Somit leiten also die Roboter eingehende Nachrichten weiter an Ihren nächsten Nachbarn bis die Nachricht des erforschenden Roboters den Empfänger erreicht hat. Das Hopping der Nachrichten ist also wesentlich effizienter als der Versand mit erhöhter Sendeleistung direkt zum Empfänger. Dabei wurde die notwendige Anzahl der Relaystationen für das Hopping minimiert. Die theoretischen Grundlagen wurden gelegt und Algorithmen entwickelt, welche in einer entworfenen Simulation (SmartS) [3] auf die Auswirkung der eingesetzten Strategien hin getestet wurden.

3.1.3 Bewertung In Phase I wurden die theoretischen Grundlagen gelegt und die angewendeten Strategien und Algorithmen auf Ihre Effizienz hin betrachtet. In Phase II wird die Ressourcenzuteilung weiter untersucht, um die Energieeffizienz und damit die „Lebenszeit“ der Roboter weiter zu maximieren. Es wird

angestrebt, dass heterogene Roboter mit unterschiedlichen Möglichkeiten zusammenarbeiten können. Dabei muss erst festgestellt werden welche globalen Aufgaben, welche Fähigkeiten in welcher Kombination benötigen. Dadurch muss die Strategie an die Vielfalt der Fähigkeiten angepasst werden. Das Projekt könnte beispielsweise eine wichtige Grundlage für zukünftige Rettungseinsätze werden. Menschenleben müssten nicht mehr aufs Spiel gesetzt werden und könnten nach erfolgreicher Suche direkt das zu rettende Individuum aufsuchen. Es gibt jedoch noch keine im praktischen Einsatz befindliche Umsetzung.

3.2 Multi-Objective Intrinsic Evolution of Embedded Systems

Dieses Projekt, geleitet von Professor Dr. Marco Platzner (Paderborn), zielt auf „Embedded Systems“ ab. Ein eingebettetes System bezeichnet ein System in das der Rechner bzw. Computer in einem technischen Kontext eingebettet ist. Das System kann dabei verteilt und autonom auf Teilsystemen laufen, was dem Organic Computing Gedanken entspricht. Die weit wichtigere Eigenschaft von „Embedded Systems“ ist jedoch, dass das System seine Steuerungs-, Regel- und Überwachungsfunktionen für den Benutzer weitestgehend transparent verrichtet [21].

3.2.1 Hintergrund und Ziele Die Vision ist es eine neu entstehende rekonfigurierbare Hardware Architektur mit Optimierungsmethoden, die der Natur entliehen sind, zu kombinieren. Damit werden neue Ansätze geschaffen mit denen es möglich ist die Selbst-Anpassung, die Selbst-Optimierung, die Robustheit und die Fehlertoleranz zu verbessern. In einem intrinsisch entwickeltem System läuft der Evolutionsprozess zusammen mit der Funktion, welche die Evolution durchlebt, auf dem gleichen Zielsystem ab und wird nicht von Außen eingesteuert. Dadurch wird das autonome Handeln des Systems gewahrt. Fügt man Rekonfigurierbare Hardware und evolutionäre Algorithmen zusammen, erhält man ein System, dass in der Lage ist seine Struktur online, also zur Laufzeit, zu reorganisieren. Zum Einen können langsame Veränderungen, wie z.B. Veränderungen in der Umgebung des System, und zum Anderen schnelle und radikale Veränderungen zu einer Rekonfiguration führen. Bei langsamen Veränderungen kann ohne Probleme ein evolutionärer Prozess angestoßen werden, wohingegen bei radikalen Veränderung auf „vor-entwickelte“ Prozesse zurückgegriffen werden muss, um das System am Laufen zu halten.

Die Ziele im Einzelnen:

- Entwicklung neuer ressourcen-bewusster Modelle und Algorithmen für die intrinsische Evoltion eines „Embedded System“
- Untersuchung von Optimierungstechniken (einer ganzen Menge an Systemen) für rekonfigurierbare Hardware
- Entwicklung der Basistechnologie für rekonfigurierbare Hardware und Software

3.2.2 Ergebnisse Als Repräsentant einer sich selbst verändernden Hardware wurde das Cartesian Genetic Program (CGP) verwendet [12]. Das CGP ist ein indizierter Graph, der Ähnlichkeit mit einem künstlichen neuronalem Netz hat. Ein Programm wird als ein solcher Graph repräsentiert, bei dem das Ganze als linearer String von Integers dargestellt wird. Die Inputs bzw. das terminal set sowie die Knoten (Outputs) sind sequentiell numeriert. Die einzelnen Knotenfunktionen sind eigenständig nummeriert. Blöcke können dabei mit AND, OR oder n_n bit table lookup kombiniert werden. Zweiter Bestandteil des Frameworks ist das graphical user interface (GUI). Für die Untersuchung gewinnt man so einen schnelleren Überblick über die Auswirkungen der Optimierung [14]. Dritter Bestandteil sind verschiedene Batch Mode Tools, welche repräsentative Durchläufe für ein Experiment erzeugen. Gleichzeitig besitzt es ein Interface zur Grid Computig Software Condor.

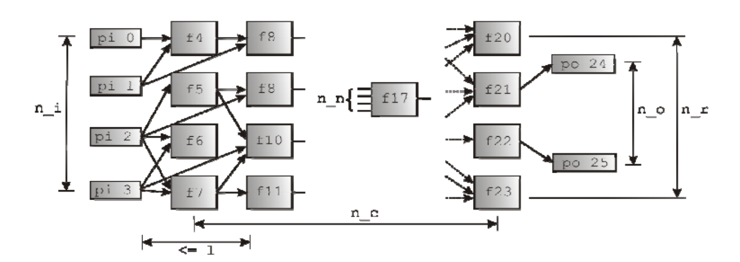


Abbildung 1. Cartesian Genetic Program [18]

Aufbauend auf dem Cartesian Genetic Program wurde das Modell für die Bedürfnisse eines „Embedded System„ angepasst und zum Embedded Cartesian Genetic Program (ECGP) erweitert. Zusätzlich wurden genetische Algorithmen implementiert und das System auf ein multi-objective System angepasst. Um das System besser testen zu können wurde ein Framework implementiert [14]. Das Framework besteht dabei aus drei Hauptbestandteilen und ist frei verfügbar. Der erste Bestandteil sind generische, evolutionäre Algorithmen Module mit deren Repräsentationsmodellen, passende Operatoren und evolutionären Algorithmen. Man startete die Implementierung eines rekonfigurierbarem System-on-Chip. In der Hardware wird das mittels eines FPGA Layouts realisiert, welches die CPU und die Logik beherbergt [13]. Für eine Fallstudie wurde die Entwicklung eines Electromyography (EMG) Signal Klassifikator begonnen [18], mit dem eine Handprothese verbessert werden soll.

3.2.3 Bewertung Mit Abschluss der Phase I wurden Modelle zur Repräsentation eines aus sich selbst heraus entwickelnden Systems untersucht. Dabei wurde der CGP Ansatz für Embedded Systems angepasst und verändert. Man entwickelte ein Framework mit dem man experimentell seine Ansätze umsetzen und kontrollieren kann [14]. Desweiteren begann man mit der Implementierung eines

self-adapting System-on-Chips. Erste Tests des Modells wurden in einer Fallstudie für einen Signal Klassifikator einer Handprothese durchgeführt. In Phase II soll auf das Skalierungsproblem eingegangen werden, die Implementierung des SoC vervollständigt und die intrinsische Implementierung des Controllers der Handprothese abgeschlossen werden. Als weitere Fallstudie für Phase II ist ein sich selbst entwickelnder Roboter Controller angedacht. Der Controller soll dann EyeBots auf einem Fussballfeld steuern und somit anderen Robotern durch seine evolutionäres Design überlegen sein.

3.3 Formal Modeling, Safety Analysis, and Verification of Organic Computing Applications (SAVE ORCA)

Gerade im Organic Computing ist Verfügbarkeit eine wichtige Anforderung, da viele Anwendungen verteilt laufen und die Selbst-X Eigenschaften erfüllt werden sollen. Das Software Design hat sich in der Vergangenheit zur Ingenieursdisziplin gewandelt. Daher müssen Grundlagen und Verfahren für den korrekten Entwurf von Software im Organic Bereich geschaffen werden.

3.3.1 Hintergrund und Ziele Es soll ein Verfahren entwickelt werden um systematisch derartige Systeme zu entwickeln. Mit einem top-down Design sollen die Systeme den Anforderungen an die Hochverfügbarkeit und die Anpassungsfähigkeit genügen. Diese Zuverlässigkeit muss dabei schon bei den Grundlagen beginnen und daher muss die Software auf funktionale Korrektheit und Sicherheit überprüft werden. Sicherheit impliziert zwei Ausprägungsformen die wir mit dem Begriff verbinden. Zum Einen Betriebssicherheit und zum Anderen die Angriffssicherheit. Diese beiden Ausprägungen sollen selbst bei unerwarteten Störungen und Fehlern weiterhin die Funktion gewährleisten können. Am Ende des Projekts soll ein Framework zur Verfügung stehen, mit dem es möglich ist ein selbstadaptives und selbstorganisierendes System zu entwickeln. Der Ansatz vereint dabei formale Spezifikation und Verifikation mit Fehleranalyse und intelligenter Rekonfiguration.

3.3.2 Ergebnisse In Phase I ist ein Organic Design Pattern (ODP) enworfen worden, mit welchem es weitaus einfacher ist Entwurfsprobleme beim Softwaredesign auszuschließen. Innerhalb des ODP gibt es verschiedene Entitäten wie z.B. agents, roles, tasks, ressources und capabilities. Dabei bildet die Rollenverteilung das Kernelement des Organic Systems. Das ODP wurde so entworfen, dass es sich sowohl für eine Implementierung eignet, als auch für eine formale Analyse [9]. Mit Hilfe des OPD wurde eine Fallstudie für einen Produktionsablauf simuliert. In dieser Produktionsumgebung gab es Roboter mit unterschiedlichen Rollen. Alle Roboter waren dabei aber gleichwertig und hatten die selben Fähigkeiten. Sollte ein Roboter ausfallen wird dies an Andere kommuniziert, die in Abstimmung untereinander ihre Rollen selbständig neu verteilen, damit der Ausfall durch eine Reparatur behoben werden kann. Dabei passen sich auch die autonomen Förderfahrzeuge der Produktionsstraße der neu verteilten Rollen und

somit an den neuen Ablauf an. Während des Ausfalles blieb der Produktionsablauf voll funktionsfähig und nach der abgeschlossenen Reparatur des defekten Robotors wurde dieser wieder in die Produktion automatisch eingebunden [9]. Es wurde ein Theorem aufgestellt mit dem sich ermitteln lässt, in welchem Grade das System selbst heilend ist, bzw. wie groß der Verlust sein kann bis das System zum Stillstand kommt. Dazu wurde das Modell in eine verification engine language übersetzt. Das dabei entstandene Ableitungsproblem lies sich automatisch lösen.

3.3.3 Bewertung Das vorliegende Design Pattern erleichtert die Entwicklung eines effizienten Organic Computing System. Dabei wurde nicht nur die Formale Verifikation des selbigen berücksichtigt, sondern auch ein formales Messinstrument für den Grad der Selbstheilung entworfen. Somit sind die ersten wichtigen Schritte gemacht für die Konstruktion eines solchen Systems. In Phase II wird das ODP in den Softwareengeneering Prozess eingegliedert, das formale Messinstrument um die anderen Selbst-X Eigenschaften erweitert, die Analyse bestehender Organic Algorithmen (Zusammenarbeit mit OCμ 3.6) und die Anwendung der verwandten Methoden auf andere Domänen (Zusammenarbeit mit ASOC 3.4) durchgeführt.

3.4 Architecture and Design Methodology for Autonomic System on Chip (ASOC)

Steigende Komplexität integrierter Schaltkreise ermöglicht immer komplexere System-on-Chip (SoC) Anwendungen. Unter SoC versteht man die Integration aller wichtigen Systemfunktionen auf einem einzigen Silizium Chip. Üblicherweise finden SoC ihr Einsatzgebiet in eingebetteten Systemen. Durch zunehmende Miniaturisierung und der damit verbundenen physikalischen Grenzen der einzelnen Transistoren wird das Fehlerverhalten jedoch immer instabiler und das System somit gegen Störung weitaus anfälliger [23]. Der Gedanke ist daher, die steigende Komplexität mit Komplexität zu bekämpfen. Dazu sollen Teile der reichlich vorhandenen Transistoren genutzt werden um OC Prinzipien zu integrieren.

3.4.1 Hintergrund und Ziele Mit diesem Konzept erhofft man sich eine höhere Fehlertoleranz, höhere Performance, bessere Energieausbeute, bessere Diagnose und die Fähigkeit umgebungsbewusst zu sein. Ein solches Konzept bedarf einer neuen frischen und ganzheitlichen Betrachtung der Chip Architektur. Eine neue Design Methode wird untersucht mit der fehleranfällige Bereiche eines Chips gesperrt und die Aufgaben auf andere Bereiche verteilt werden können.

3.4.2 Ergebnisse Zunächst wurden mögliche Fehlerarten identifiziert und Ansätze entwickelt wie auf diese zu reagieren ist. Für die Bewältigung dieser Aufgaben wurde das automic layer in die SoC Architektur integriert. Dieser

neue Layer impliziert Flexibilität, Lernen, erlaubt Selbst-Heilung und ermöglicht zusätzlich eine systemweite Optimierung. Da sich die Probleme aber in zwei Klassen einteilen lassen wurde auf der Hardwareseite der Autonomic Layer eingeführt, gleichzeitig wurde auf der Softwareseite eine Design Methodology geschaffen, mit der schon beim Entwurf von Anwendungen Fehler vermieden werden sollen.

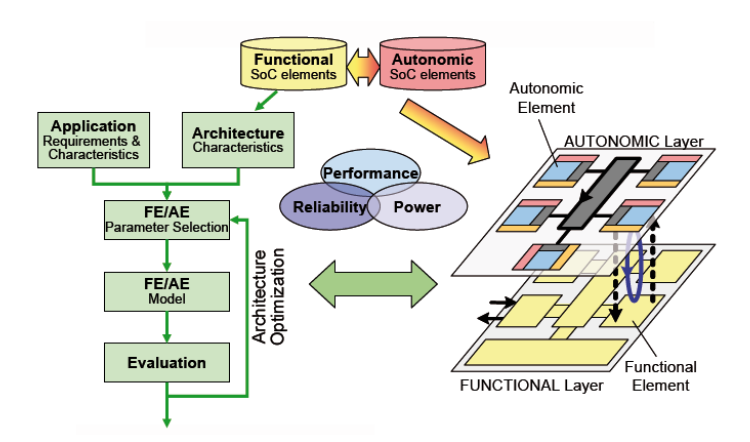


Abbildung 2. ASOC Design Methodologie und Autonomic Hardware Layer [23]

Am Ende von Phase I steht nun neben einer Design Methode auch ein Hardware Modell zur Verfügung. Darüber hinaus existiert eine Selbstheilende CPU Pipeline, die fehlertolerant ist. Ein Selbstheilender BUS wurde ebenfalls entwickelt.

3.4.3 Bewertung Für Phase II werden Monitore entwickelt um eine ausgeglichene Zuverlässigkeit der Performance gewährleisten zu können, damit der Chip auch im Fehlerfall noch mit angemessener Performance weiter arbeiten kann. Konzepte für die Hardware und Software Implementierung sollen geschaffen werden und die einzelnen Autonomic Elements (AE) des Autonomic Layers verbunden werden, um einen emergenten Mehrwert zu bilden. Endziel ist es, einen ASOC Simulator und einen Prototyp zu erzeugen. Es gibt viele Einsatzgebiete für eine derartige fehlertollerante Architektur. Beispielsweise wenn einzelne Bereiche des Chips nicht mehr funktionieren oder nur eingeschränkt bzw. langsam anzusprechen sind. Der ASOC Chip könnte diese Bereiche umlagern und somit eine maximale Fehlertoleranz erreichen. Interessant wäre ein derartiger Chip nicht nur für den Organic Computing Bereich, sondern auch für viele weitere Anwendungsgebiete in denen Selbst-Heilung und Fehlertoleranz unabdingbar sind. ASOC ist ein vielversprechendes Konzept und erhält, mit zunehmendem Grad der physikalischen Transistorgrenzen, immer größere Bedeutung.

3.5 Self-organized and self-regulation coordination of large swarm of self-navigating autonomous vehicles, as occuring in highway traffic (AutoNomos)

In diesem Projekt beschäftigt man sich mit einem verteilten und selbstregulierenden Ansatz zur Selbstorganisation von großen Systemen mit mobilen Objekten. Ein passendes Anwendungsszenario ist dabei der Straßenverkehr. Im Moment wird die Verkehssituation durch Induktionsschleifen im Straßenbelag erfasst und an eine zentrale Verkehrsleitstelle propagiert. Von dort aus werden aus den Daten wichtige Informationen, wie Stausituation, generiert und an die Verkehrsteilnehmer weitergeleitet. Die Weiterleitung funktioniert dabei im Moment entweder per Ferkehrsfunk bzw. TMC oder GSM bzw. GPRS. Basierend auf Methoden für mobile ad-hoc Netzwerke und Ideen aus dem Bereich verteilter Algorithmen soll die Erkennung optimiert werden und Vorhersagen über den Verkehrfluss möglich sein.

3.5.1 Hintergrund und Ziele Die Herausforderung die sich dabei stellen sind nicht nur monetärer sondern auch „politischer“ Natur: Zu klären gilt es, wer für die Montage und Instandhaltung der Sensoren aufkommt und wer die gewonnen Informationen verwaltet. Dabei muss natürlich beachtet werden, dass nicht überall Sensoren installiert werden können. Gewonnene Informationen sollen dabei eine Korrektheit von min. 70% gewährleisten können. Eine günstige Gelegenheit bietet dabei die fortschreitende Technologie der Fahrzeuge auf den Straßen. Die Fahrzeuge vereinen mehr Kommunikationsmöglichkeiten in sich als je zuvor. Car2Car Kommunikation beschreibt dabei einen Ansatz wie Fahrzeuge untereinander Informationen und Wissen austauschen können [16]. Dieses Projekt verwendet diesen Ansatz um auf komplexe, sich verändernde Verkehrssituationen angemessen reagieren zu können. Als übergeordnete Ziele sollen dabei der Gesamtbenzinverbrauch und die mittlere Reisezeit gesenkt werden.

3.5.2 Ergebnisse Statt Induktionsschleifen in der Fahrbahn werden schwebende Datenwolken (Hovering Data Clouds , HDCs) an Verkehrsstrukturen, z.B. Verkehrsstaus, etabliert die durch ein Ereignis hervor gerufen werden [5]. Diese bilden organische Informationseinheiten (Organic Information Complexes, OICs) die als funktionale Einheiten den Verkehrsfluss beschreiben. Die HDCs werden wiederum von einzelnen Fahrzeugen „getragen“, können sich aber unabhängig vom Fahrzeug bewegen und verändern. Somit kann mittels am Stauende und -anfang aufgebauter HDCs durch einen OIC ein Verkehrsstau erkannt werden, selbst wenn die im Stau befindlichen Fahrzeuge wechseln. Einzelne Fahrzeuge leiten ihre Informationen an andere weiter und somit bleiben die HDC´s am Stauende bzw -anfang. Der Datenaustausch funktioniert mittels 802.11 Standard. Damit ist ein Radius von 250m bei 1Mbit/s Übertragungsleistung möglich. Es wurden vier verschiende Übertragungsprotokolle untersucht die in diesem Szenario eingesetzt werden könnten. Nach der Untersuchung der Auslieferungsrate der Nachrichten und der Geschwindigkeit bei maximalem Datenverkehrsaufkommen erwies sich das entwickelte AutoCast Protokoll als bester Kompromiss [22].

AutoCast funktioniert wie wahrscheinlichkeitstheoretisches Fluten des Netzes, periodisch wird dabei eine vollständig Liste mit Hash Werten der Daten Einheiten per Broadcast verteilt. Wenn eine unvollständige Liste empfangen wird muss sofort mit gleicher Wahrscheinlichkeit geantwortet werden. Damit ist es möglich selbst bei maximaler Netzauslastung noch kommunizieren zu können. Für die Simulation wird das selbst entwickelte Tool SUMO eingesetzt. Zusätzlich wurden die Erkenntnisse real mit 10 Robotern, der Firma Lego, exemplarisch getestet.

3.5.3 Bewertung Das Projekt verfolgt einen interessanten und neuen Ansatz zur Stauerkennung und Vorhersage. Allerdings ist es sehr stark von der technischen Grundlage, der Car2Car Kommunikation, abhängig. Dort gibt es noch viele Unklarheiten über Sicherheit und Zuverlässigkeit des 802.11 Übertragungsmedium. Der praktische Einsatz ist dabei einerseits von der Akzeptanz der Fahrzeugführer zu Car2Car und der Akzeptanz der Verkehrsleitstellenbetreiber abhängig. Dennoch wird das Projekt in Phase II fortgeführt mit dem Fokus auf die Struktur der Verkehrsnetze und auf Algorithmen für das distributed computing. Die Zusammenarbeit mit anderen Projekten im SPP1183 wird dabei noch intensiviert werden.

3.6 Organic Computing Middleware for Ubiquitous Environments

Mit diesem Teilprojekt soll eine Organic Computing Middleware geschaffen werden (OCμ). Geboren wurde die Idee aus einem weiteren Projekt des Lehrstuhls, dem „smart doorplate project“ [24]. Die Doorplates sind Bestandteil des „flexible office“ Konzepts. Man will durch interaktive Türschilder dem Besucher den Weg weisen oder z.B. Informationen über die Belegung eines Raumes über diese door plates verbreiten. Hierbei wurde eine spezialisierte Middleware eingesetzt, die Vision ist es aber eine universell einsetzbare Middleware zu schaffen.

3.6.1 Hintergrund und Ziele Eine universelle Organic Computing Middleware soll entworfen werden, bei der das System auf vielen Knoten (nodes) verteilt läuft. Eine Applikation ist dann eine Bündelung von Services, die auf den einzelnen Knoten laufen. Während der Laufzeit können diese Services umgezogen werden, falls ein Knoten überlastet ist oder die Gefahr besteht, dass dieser bald ausfällt. Dabei sollte sich das Gesamtsystem, in Abhängigkeit einiger vorgegebener Paramter selbst optimieren. Ein weiteres Designziel war es, ein asynchrones eventbasiertes Nachrichtensystem zu schaffen, bei dem der Nachrichtenaustausch mit getypten Nachrichten verläuft.

3.6.2 Ergebnisse Wie in Abbildung 3 zu sehen wurde bei OCμ eine knotenbasierte 2-Schicht Architektur geschaffen. Die Organic Computing Prinzipien, wie die Selbst-X Eigenschaften wurden als Services bzw. Monitors implementiert. Der Transport selbst wird per p2p mit JXTA umgesetzt. Der Event Dispatcher

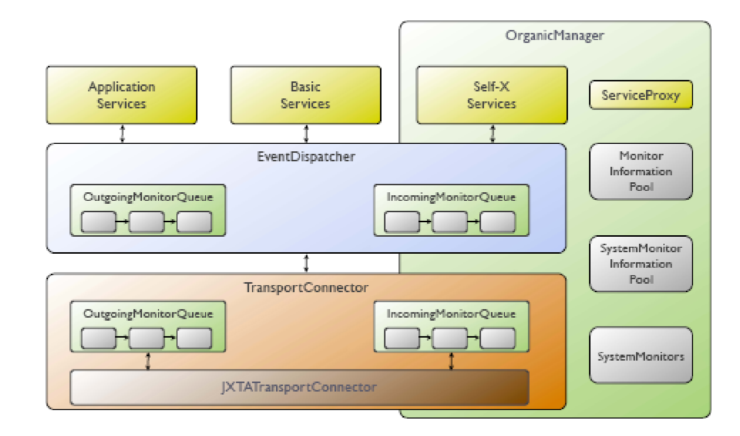


Abbildung 3. OCμ Middleware Architektur [25]

kümmert sich um die Nachrichtenlieferung. Sowohl Broadcast Mitteilungen als auch direkt adressierte Nachrichten werden unterstützt. Der Event Dispatcher entscheidet, ob die Nachrichten lokal zugestellt werden können, oder ob diese an einen anderen Node geschickt werden müssen. Der Service Proxy kümmert sich automatisch um die Nachrichtenzustellung für Services die mitlerweile auf einen andern Node umgezogen sind. Durch die optimale Verteilung der einzelnen Services auf unterschiedliche Nodes hat man eine gute Skalierbarkeit erreicht. Monitore sammeln die versandten Informationen über den Ist-Zustand der Knoten für den Organic Manager, welcher entscheidet ob ein Service evtl. umgezogen werden sollte. Im Bereich der Selbstheilung wurde ein „distributed self-healing data store“ geschaffen in dem die Statusinformationen der einzelnen Services abgelegt werden. Die Fehlererkennung überwacht die Kommunikationsfähigkeit der Services, um evtl. nicht errreichbare Services bzw. Nodes zu detektieren. Wurde ein Ausfall erkannt, wird versucht den Service neu zu starten oder auf einen anderen Knoten umgezogen. In Anlehnung an den menschlichen Körper gibt es Antikörper und einen Thymus, der Antikörper mittels Negativselektion der Bitmuster der Nachrichten auswählt. Übrig bleiben damit nur Antikörper die auf bösartige Muster hin deuten und zu deren Detektion dienen. Umgebungsbewusstsein und vorrauschauendes Handeln ist Hauptaufgabe der Applikation die dann mit OCμ realisiert wird. Die Middleware bildet nur einen Vermittler.

3.6.3 Bewertung Die im Rahmen dieses Projektes generierte Middleware wird momentan auch in dem „Smart door plate project“ [24] an der Universität Augsburg eingesetzt. In Phase II des Projektes soll die Selbst-Konfiguration verbessert werden. In diesem Bereich möchte man auch mit dem Project SAVE ORCA 3.3 zusammenarbeiten und die Selbst-Konfiguration formal verifizieren. Die Fehlererkennung soll nicht nur weiter verbessert werden, sondern darüber hinaus soll eine Fehlerwiederherstellung eingeführt werden die sich an psycholo-

gischen und soziologischen Prinzipien orientiert. Durch einen dezentralen Thymus möchte man auch den Selbstschutz perfektionieren und komplettieren.

3.7 Energy Aware Self Organized Communication in Complex Networks

Das Projekt der Universität Rostock beschäftigt sich mit der Erforschung der Regeln und des Verhaltens mit denen es die Natur schafft aus diesen einfachen Mustern komplexe Systeme bzw. Netzwerke zu steuern.

3.7.1 Hintergrund und Ziele Der Fokus wird hierbei auf komplexe Netzwerkstrukturen gelegt. Ziel ist es den Zusammenhang zwischen Ursache und Effekt in komplexen skalierungsfreien Systemen zu behandeln. Als Musterbeispiel einer solchen Struktur wurden Sensornetze gewählt. Ein Sensornetz ist dabei ein drahtlos kommunizierender adhoc Verbund winziger Computer die mittels ihrer Sensoren eine Umgebung überwachen. Charakteristikum solcher Sensornetze ist die hohe Anzahl an Mitgliedern und die beschränkte Kommunikationsreichweit und Rechenleistung. Für ständig ändernde Gegebenheiten und Fehlertolleranz muss das Sensornetz gewappnet sein. Die von jedem einzelnen Knoten gesammelten Daten werden dann an den Empfänger (Senke) weitergeleitet. Konkrete Anwendungsszenarien können dabei z.B. Brandschutzbekämpfung, Eruptionsüberwachung oder der Schutz vor Überflutung sein. Das Ziel ist es, unter der Verwendung der Organic Computing Prinzipien die Funktion des Netzes maximal lange aufrecht erhalten zu können und die Robustheit zu maximieren. Heutige Netze wie z.B. das Internet besitzen wenige Knotenpunkte mit vielen Verbindungen und sind somit fehleranfälliger für Angriffe als Netze die aus vielen Knoten mit wenig Verbindngen bestehen.

3.7.2 Ergebnisse Jeder einzelne Netzknoten kann dabei selbst seine hierachisch aufgebaute Rolle ändern und sich auf eine Aufgabe spezialisieren, ohne jedoch das Gesamtziel zu verletzen. Die Lebenszeit des Netzes konnte mit dynamischer Rollenanpassung um 40% gegenüber der einmalig zugeteilten Rollen verlängert werden. Eine virtuelle Partitionierung des Netzes in Zellen diente dem Ziel, dass jeder Knoten in Reichweite die Rollen des andern übernehmen kann ohne dabei abhängig von der Position zu sein. Ein Verhalten was auch von Schwärmen in der Natur bekannt ist. Damit können alle überflüsigen Knoten deaktiviert werden und somit Energie sparen solange ein Knoten eine Zelle abdecken kann. Es verbleiben die Knoten die bei einem Ausfall die Funktion des Netzes gewährleisten können. Dafür wird zuerst von der Senke aus ein Netzwerk aufgebaut so das jeder Punkt von überall erreichbar ist und das Netz ein skalierungsfreies Verhalten aufweist. Danach wird das Netz optimiert um die Funktion mit minimalen „Kosten“ zu gewährleisten. Das skalierungsfrei Netz bracht in den Untersuchungen eine Erhöhung der Lebenszeit von 130% gegenüber konventiellen Netzen [11].

3.7.3 Bewertung In Phase I des SPP1183 wurden durch die Untersuchungen die Robustheit und die Lebenszeit des Verbundes maximiert. Die Verbesserung ist dabei die Summe der einzelnen Optimierungen im Schwarmverhalten und in der Rollenzuteilung durch das skalierungsfreie Netz. In Phase II wird der Fokus auf die Anpassung der Netzstrukturen auf unvorhergesehene Fälle gelegt, um die aufgetragene Aufgabe auch unter minimalem Energiebedarf durchführen zu können. Das Projekt hat die Grundlagen untersucht und sich an Prinzipien der Natur orientiert. Die gewonnen Grundlagen können auch in vielen anderen Anwendungsgebieten verwendet werden. Jedoch bietet allein das Feld der Sensornetze breite Anwendungsspektren.

4 Projekte im Fokus

Im folgenden Kapitel wird auf zwei ausgewählte Projekte näher eingegangen.

4.1 Organic Traffic Control (OTC)

Sie Stausituation auf den Straßen wird immer schlechter. Die Kapazität der Straßen reicht in Stoßzeiten oftmals nicht aus, um den Verkehrsfluss nicht ins Stocken geraten zu lassen. Genau diesen Sachverhalt möchte man verbessern. Dieses Projekt ist eine Kooperation von der Universität Karlsruhe und der Universität Hannover. Das Projekt verwirklicht dabei ein dezentrales, adaptives und selbst-lernendes Systeme um den Verkehrsfluss bestehender Infrastrukturen zu maximieren.

4.1.1 Hintergrund und Ziele Heutzutage verfügen Verkehrsknotenpunkte über moderne Schaltanlagen für die Ampelsteuerung. Die Ampelsteuerungen werden dabei aber an Hand von manuellen Datenerhebungen optimiert und auf fixe Schaltzeiten eingestellt. Dabei kann zwischen mehreren Schaltzeit-Programmen gewählt werden. Diese Auswahl ist aber fix anhand der jeweiligen Tageszeit und des Datums gesetzt. Es werden die Schaltzeiten zu Stosszeiten entweder verkürzt oder verlängert. Die Motivation ist diese starre Verhaltensweise durch ein autonomes System zu ersetzen und die Möglichkeiten bzw. die Grenzen eines solchen dezentralen und adaptivem Ansatz zu beleuchten. Ziel in Phase I ist es eine generische und modulare Architektur zu schaffen und diese in einen einzelnen isolierten Knotenpunkt zu implementieren. Später soll dann die fexible Zusammenarbeit mit weiteren Kntotenpunkten erfolgen. Das System muss sich dabei an drei Vorkommnisse anpassen:

- Langzeitveränderungen
- Kurzzeitveränderungen
- Wiederverwendung des Gelernten

Die längerfristigen Veränderungen sollen durch kontinuierliches Lernen des Systems erreicht werden, um damit die Verhaltensweisen zu optimieren. Unter Kurzzeitveränderungen versteht man Schwankungen in der Verkehrslast oder Ausfälle. Bei einem solchen Vorkommnis muss schnell gehandelt werden. Dies ist mit einem Reflex in der Natur vergleichbar. Gewisse Aufgaben müssen im Fehlerfall sofort durchgeführt werden. Das System soll aber auch das bisher Gelernte wieder verwenden können. Um dies möglich zu machen, ist ein Speicher für das Wissen notwendig auf den das System dann zurück greifen kann. Eine Ampel muss ein hochverfügbares System sein, sie muss dabei aber extrem fehlertollerant und exakt arbeiten. Ansonsten wäre der sichere Verkehrsfluss auf einer Kreuzung nicht mehr gewährleistet. Deshalb steht die Sicherheit in diesem Projekt ganz groß geschrieben. Um den Effekt möglicher Fehler feststellen zu können werden die verschiedenen Szenarien simuliert, da diese nicht real getestet werden können. Alle Änderungen und Anpassungen müssen dabei aber nachvollziehbar bleiben. Das bedeutet nichts anderes, als dass es für den Menschen lesbar bleiben muss. Dabei soll aber der manuelle Eingriff so gering wie möglich bleiben um die Autonomie des Systems gewährleisten zu können.

4.1.2 OTC Architektur Die Ziele die sich gesetzt worden sind erfodern eine Multilayer Architektur des Organic Traffic Control Systems. In diesem Fall entschied man sich für eine 3-Schicht Architektur.

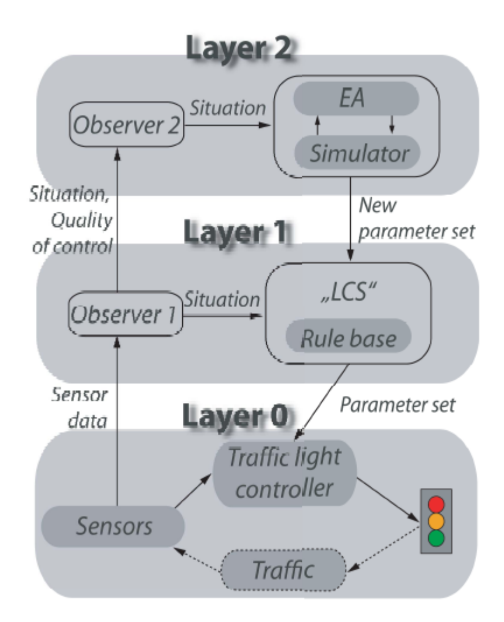


Abbildung 4. Organic Traffic Control 3-Schicht Architektur [2]

Layer 0 bildet die Steuerung der Ampel, diese erhält Sensordaten und schaltet dann durch den Controller die entsprechenden Lichtsignale. Der Verkehr wird dadurch beeinflusst und liefert wiederum neue Sensordaten. Layer 0 kann aber nur auf kleine Veränderungen im Verkehr reagieren. Das Verhalten wird dabei von einigen wenigen festen Parametern bestimmt. Layer 1 und 2 sind wieder nach dem bekannten Observer-Controller Prinzip aufgebaut. Layer1 beherbergt den Observer 1, der die Verkehrssituation überwacht und versucht mit Hilfe eines modifiziertem Learning Classifier System (LCS) anhand passender Regeln ein angemessenes Parameter-Set auszuwählen. Es werden also in Layer 1 größere Veränderungen im Verkehr behandelt, denn genau diese größeren Veränderungen benötigen ein neues Paramter-Set. Layer 3 ist für die eigentliche Erzeugung von passenden Parametern zuständig. Dazu werden diese mittels eines Evolutionären Algorithmus (EA) erzeugt und deren Qualität für einen praktischen Einsatz in einer Simulationssoftware getestet. Layer 2 kann damit auch auf ein externes System ausgelagert werden. In Phase I liegt das Augenmerk hier auf der Selbst-Parametrisierung, erst in Phase II will man dann an der Kooperation verschiedener Traffic Light Controller (TLC) arbeiten. Das LCS erhält als Eingabedaten die Fahrzeuge pro Stunde (***flow***) in Relation zu diesem Knotenpunkt. Als Ausgabe wird ein Parameter-Set, welches das Programm für den TLC beschreibt, erwartet. Das Set kann dabei entweder eine fixe oder eine flexible Zeitsteuerung der Ampel repräsentieren. Die zu optimierende Funktion ist dabei der Level of Service (LOS). Aus dem in Amerika gebräuchlichen Highway Capacity Manual (HCM) ist dabei dieses Konzept der Unterteilung der Verkehrsqualität in Stufen in das deutsche Handbuch für die Bemessung von Straßenverkehrsanlagen übernommen worden. Es spiegelt die durchschnittliche Wartezeit pro Verkehrsteilnehmer wieder, was wiederum Rückschlüsse auf die Qualität und die Leistungsfähigkeit der Verkehrstechnischen Einrichtung zulässt. Die Wartezeit soll dabei minimiert werden [2]:

$$LOS = \frac{\sum flow * t_{delay}}{\sum flow}$$

Für die Simulation wurden große Kreuzungsbereiche in Hamburg und Hannover verwendet. Von diesen Knotenpunkten liegen Verkehrszählungen und die jetzigen Ampelschaltpläne vor. Nun implementierte man diese Kreuzungen um damit am PC testen zu können. Man muss erst offline Lernen um ein Verständnis für die Auswirkungen der verschiedenen Parameter zu entwickeln. Die Veränderungen werden dann ermittelt und mit dem vorhanden starrem, zeitbasiertem Referenzcontroller verglichen. Dabei ist vorallem die Reaktionszeit und die positive Auswirkung auf den Verkehrsfluss interessant.

4.1.3 Ausblick und Bewertung In Phase II soll nun die Zusammenarbeit verschiedener TLC getestet werden. Es bieten sich dafür zwei mögliche Ansätze an. Zum einen eine hierachisch aufgebaute Organisation mit zentraler Steuerung zum anderen eine dezentral organisierte Struktur. In der zentralen Steuerung

überwacht ein einzelner Observer das gesamte Netzwerk an TLCs und nimmt evtl. Änderungen vor. Somit beeinflussen seine Änderung, die das gesamte Netzwerk optimieren sollen, auch die Strategien der einzelnen Knoten. Anders verhält es sich beim dezentralen Ansatz: Dort müssen die einzelnen Knoten Ihre Änderungen und ihre Daten an die Nachbarknoten weiterleiten. Diese können dann entscheiden wie sie darauf reagieren, um das gesamte Netz zu optimieren.

Am Ende von Phase I liegt nun die fertige Architektur vor und eine Simulation für die Tests wurde implementiert. Beispielsweise konnte die durchschnitliche Wartezeit von 61,2s beim Referenzcontroller auf 36,3s gesenkt werden [2]. Für Phase II des SPP wird eine intensive Zusammenarbeit mit dem Projekt *„Quantitative Emergence (QE) - Metrics, Observation and Control Tools for Complex Organic Ensembles“* *4.2* angestrebt. Das emergente Verhalten des Systems soll somit verbessert werden. Wie auch schon bei den anderen Projekten wurden zahlreiche Veröffentlichungen publiziert. Das Projekt wird in Phase II unter dem Begriff OTC^2 weiter geführt. Man darf also gespannt sein, inwieweit sich dieses innovative und vielversprechende Projekt entwickelt.

4.2 Quantitative Emergence (QE) - Metrics, Observation and Control Tools for Complex Organic Ensembles

Aufgrund zunehmender Komplexität der einzelnen Computer Systeme und deren stetig anwachsende Zahl ist es nötig geworden nach neuen Konzepten zu suchen, die diese beherrschar machen. Um gegen Angriffe, Ausfälle und anderen unerwarteten Geschehnissen bestehen zu können setzt man auf die Selbst-X Eigenschaften, vor allem aber auf die Selbstorganisation. Für dieses Projekt zeichnet Professor Dr. Hartmut Schmeck (Karlsruhe) verantwortlich in Zusammenarbeit mit Professor Dr.-Ing. Christian Müller-Schloer (Hannover) und Dr. Jürgen Branke (Karlsruhe). Das Hauptaugenmerk wird hierbei auf die Emergenz gelegt, diese kann mit dem Grundsatz´„Das Ganze ist mehr als die Summer seiner Einzelteile“ skizziert werden, was wiederum auch unter dem Begriff der Schwarmintelligenz zusammengefasst werden kann.

4.2.1 Hintergrund und Ziele In Phase I hat man sich im Groben drei Ziele gesezt.

– Die Untersuchung und die genaue Betrachtung des Ablaufs von Emergenz in selbst-organisierenden Systemen

  Dieses Ziel soll hauptsächlich in Hannover untersucht werden.

– Die Untersuchung und die genaue Betrachtung der Mechanismen um die Emergenz zu beeinflussen bzw. deren Effekte zu steuern

  Dieses Ziel soll hauptsächlich in Karlsruhe verwirklicht werden.
  und als drittes Ziel:

– Die Realisation von Tools und deren Validierung in verschiedenen Test Szenarien

Für Phase I wird man von konkreten technischen Anwendungen abstrahieren und einfachere künstliche Systeme untersuchen. Erst später will man dann seine Erkenntnisse mit dem Projekt Organic Traffic Control (4.1) zusammenbringen.

4.2.2 Architektur und Strategien Man hat eine Observer Controller Architektur geschaffen um autonome Teile zu koordinieren und ein emergentes Verhalten entstehen zu lassen. Als Beispiel diente ein Schwarm Hühner. Jedes Huhn mit seiner eigenen „beschränkten" Intelligenz sollte zusammen mit den Anderen einen intelligenten Schwarm bilden. Dazu wurde zuerst ein Schwarm Hühner implementiert der anschließend mit der Observer Controller Architektur überwacht wurde. Der Observer überwacht dabei das Verhalten des Schwarms und leitet den gemessenen Kontext an den Controller weiter. Nun leitet der Controller, in Abstimmung mit einem extern gegebenem Ziel, angemessene Änderungen des Systems ein. Integriert wurde hiebei auch ein Verfahren des maschinellen Lernens (XCS)[6].

Aufzugsteuerungen dienten neben dem Hühnerschwarm zusätzlich als Testumgebung. So wurden mit der geschaffenen SuOC (System under observation and control) Architektur 50 Durchläufe in einem Gebäude mit zehn Stockwerken und vier Aufzügen untersucht und optimiert. Ein Durchlauf hatte dabei 10.000 Zeiteinheiten. Das Ganze wurde, wie auch schon die Hühner, Agentenbasiert implementiert. Für die Kontrolle der Aufzüge bildeten sich folgende zwei adequate Strategien heraus [6]:

- **singleCarBlind:**
  - Wähle den Lift mit größtem Abstand gegenüber dem Ziel zum nächst möglichen
  - Dieser Lift reagiert nicht mehr auf Erdgeschoss-Rufe
  - Der ausgewählte Lift wird beschleunigt

- **nextCallHide:**
  - Wähle den Lift mit größerem Abstand zum Ziel zu nächst möglichen Liften
  - Alle Lifte reagieren nicht auf den nächsten Erdgeschoss-Ruf
  - Alle ausgewählten Lifte werden beschleunigt

Es zeigte sich, dass die singleCarBlind Strategie in allen durchgeführten Tests den besten Kompromiss darstellte und sich die durchschnittliche Wartezeit damit um fast 35% verkürzen ließ [6].

4.2.3 Ausblick und Bewertung Mit Abschluss der Phase I steht eine Observer Controller Architektur zur Verfügung, in die bereits maschinelles Lernen integriert wurde. Durch experimentelle Ermittlung wurden die Auswirkungen und das Verhalten von Emergenz ermittelt. In den einfacheren beispielhaften

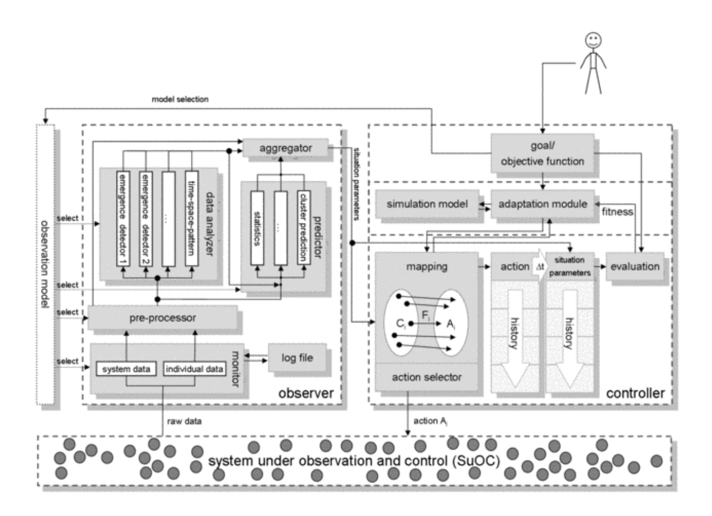


Abbildung 5. SuOC Architektur [7]

Anwedungsszenarien ging es vorwiegend um das Lernen eines Einzelnen. In Phase II soll dies nun durch das kollektive Lernen erweitert werden und die gesamte Observer Controller Architektur auf verteilten System ablaufen. Aus den dabei gewonnenen Erkenntnissen erhofft man sich neue design patterns für den Entwurf neuer Systeme. Als komplexes Anwendungsgebiet wird für die Phase II eine Zusammenarbeit mit dem OTC Projekt angedacht (4.1).

5 Zusammenfassung und Ausblick

In dem SPP1183 wurden die unterschiedlichsten Projekte durchgeführt von denen hier neun ausgewählte vorgestellt wurden. Die Auswahl fiel dabei auf Projekte, die entweder einen neuen Ansatz verwenden oder die Ergebnisse in interessanten Anwendungsszenarios umsetzen.

5.1 Beurteilung der Ergebnisse

Viele der vorgestellten Themen befassen sich mit der Erforschung der Grundlagen und dem grundsätzlichen Verständnis über die Verfahren der Natur. Gleichzeitig versucht man dies in aktuellen Anwendungsszenarien praktisch einzusetzen. Wichtig ist dabei die Zusammenarbeit und Abstimmung der Erkenntnisse untereinander. Manche Projekte verfolgen dabei einen ähnlichen Ansatz und können durch die Bündelung ihrer Kompetenzen besser agieren. Durch das Schwerpunktprogramms sollen die in 5 bis 8 Jahren verfügbaren technologischen Komponenten die Basis für den Systementwurf darstellen. Da die zukünftigen Computersysteme immer komplexer in ihrer eigenen Struktur und der Struktur ihrer Kommunikation werden, wird es zunehmend schwerer für ein Individuum

die gesamte Struktur zu überblicken und zu verstehen. Durch das Organic Computing sollen zukünftige Systeme selbstständig interagieren und kommunizieren und dabei sehr fehlertollerant und zuverlässig die Arbeit verrichten ohne von einer zentralen Stelle gesteuert zu werden. Dennoch muss dabei sicher gestellt sein, dass in letzter Instanz eingegriffen werden kann und das System nicht den Anwender dominiert.

5.2 Stand der Forschung

Zunehmende Vernetzung der gegenwärtigen Geräte und Computer erhöht den Druck diese beherrschbar und kontrollierbar zu machen. Die Grundlagen des Organic Computing wurden an die Natur angelehnt und auf die Technik projeziert. Dennoch befindet sich der Forschungszweig des Organic Computing noch immer in den Anfängen. Der seit 2002 bestehende Zweig der Informatik etablierte sich rasant, da er schnell in der Lage war Fragestellungen beantworten zu können die teilweise bereits vor vielen Jahren formuliert wurden. Dennoch wurden schon vor 2002 Teile der Organic Computing Prinzipien, wie Selbstorganisation behandelt, wenn auch in einem anderen Gesamtkontext. Zugute kam dem Organic Computing das immer noch geltende „Moore ́s Law", denn mittlerweile ist ein Stand der Entwicklung erreicht, bei dem komplexe Rechenoperationen, verschiedenste Sensoren und Kommunikationsmöglichkeiten auf kleinstem Raum kostengünstig produziert und vertrieben werden. Bis 2012 werden Chips mehrere Milliarden Transistoren enthalten und damit noch komplexere Anwendungen ermöglichen [20]. Um von Beginn an die Entwicklung und Forschung besser steuern zu können hat die IEEE eine Taskforce Emergent Technologies Technical Committee *ETTC* gegründet um die Erkenntnisse neuer Ansätze geordnet zu kontrollieren und Normen zu schaffen.

5.3 Ausblick (Phase II)

In Phase II des SPP1183, die im Juli 2007 begonnen hat, werden die grundlegenden Untersuchungen zu Prinzipien selbstorganisierender Systeme vertieft, da fast alle Projekte in Phase I die Grundlagen gelegt haben mit denen die experimentelle Erprobung generischer Architekturkonzepte und Werkzeuge erst möglich sind. Die Zusammenarbeit der einzelnen Projekte untereinander wird dabei weiter intensiviert und die gesetzten Ziele angepasst.

Literatur

[1] Christian Müller-Schloer, Christoph von der Malsburg, Rolf P.Würtz. Organic computing. *Informatik Spektrum*, 27(4):2–6, August 2004.

[2] Christian Müller-Schloer, Hartmut Schmeck, Jürgen Branke, Jörg Hähner. Organic traffic control 5th spp1183 colloquium at university of lübeck. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/inhalte/FilesColloquiumLuebeck/FilesColloquiumLuebeck/21.pdf, September 2007.

[3] M. Dynia, A. Kumlehn, J. Kutylowski, F. Meyer auf der Heide, and C. Schindelhauer. Smarts simulator design. manual, jan 2006.
[4] F. Heylighen and C. Gershenson. The meaning of self-organization in computing. http://pespmc1.vub.ac.be/Papers/IEEE.Self-organization.pdf, Juni 2003.
[5] Fekete, Sandor P., Schmidt, Christiane, Wegener, Axel and Fischer, Stefan. Hovering data clouds for recognizing traffic jams. 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, 2006.
[6] Hartmut Schmeck, Christian Müller-Schloer, Jürgen Branke, Jörg Hähner. Quantitative emergence - metrics, observation and control tools for complex organic ensembles 5th spp1183 colloquium at university of lübeck. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/inhalte/FilesColloquiumLuebeck/FilesColloquiumLuebeck/20.pdf, September 2007.
[7] Hartmut Schmeck, Christian Müller-Schloer, Jürgen Branke, Jörg Hähner. Quantitative emergence poster 4th spp1183 colloquium. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/inhalte/poster/10.pdf, Februar 2007.
[8] Hartmut Schmeck, Lei Liu. Organic computing initiative SPP1183 Project Website. http://www.organic-computing.de/spp, 2007.
[9] Hella Seebach, Frank Ortmeier, Wolfgang Reif, editor. *Design and Construction of Organic Computing Systems*. 2007 IEEE Congress on Evolutionary Computation, 2007.
[10] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology, 2001.
[11] Jakob Salzmann and Stephan Kubisch and Frank Reichenbach and Dirk Timmermann. Energy and Coverage Aware Routing Algorithm in Self-Organized Sensor Networks. In *Fourth International Conference on Networked Sensing Systems*, pages 77–80, Juni 2007.
[12] Julian F. Miller, Peter Thomson, editor. *Cartesian Genetic Programming*. EuroGP2000, Springer-Verlag, 2000.
[13] P. Kaufmann and M. Platzner. Multi-objective intrinsic hardware evolution. In *Proceedings of the 2006 MAPLD International Conference*, Washington D.C., USA, September 2006.
[14] P. Kaufmann and M. Platzner. Moves: A modular framework for hardware evolution. In *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 447–454, Washington, DC, USA, 2007. IEEE Computer Society.
[15] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.
[16] T. Kosch, editor. *Technical Concept and Prerequisites of Car2Car Communication*. 5th European Congress and Exhibition on ITS, Juni 2005.
[17] C. Müller-Schloer. Organic computing - on the feasibility of controlled emergence. In *CODES+ISSS '04: Proceedings of the international conference on Hardware/-Software Codesign and System Synthesis*, pages 2–5, Washington, DC, USA, 2004. IEEE Computer Society.
[18] Paul Kaufmann, Marco Platzner. Multi-objective intrinsic evolution of embedded systems (moves) project website 5th spp1183 colloquium at university of lübeck. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/inhalte/FilesColloquiumLuebeck/FilesColloquiumLuebeck/10.pdf, September 2007.

[19] C. C. O. Prof. Dr. Christian Schindelhauer. Smart teams: local distributed strategies for self-organizing robotic exploration teams 5th spp1183 colloquium at university of lübeck. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/inhalte/FilesColloquiumLuebeck/FilesColloquiumLuebeck/13.pdf, September 2007.

[20] Prof. Dr. Hartmut Schmeck, Prof. Dr.-Ing. Christian Müller-Schloer, Prof. Dr. Theo Ungerer. Antrag auf Einrichtung eines neuen DFG - Schwerpunktprogramms - Organic Computing. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/download/SPP-AntragOC20040216.pdf, Februar 2004.

[21] Prof. Dr.-Ing. Ch. Ament. Modellbasierter Entwurf von Steuerungen und Regelungen. http://www.imtek.de/systemtheorie/content/upload/vorlesung/2005/steuerungsentwurf-ss2005-01.pdf, 2005.

[22] Wegener, Axel, Hellbrück, Horst, Fischer, Stefan, Schmidt, Christiane and Fekete, Sandor. Autocast: An adaptive data dissemination protocol for traffic information. Proceedings of the 66th IEEE Vehicular Technology Conference Fall, 2007.

[23] Wolfgang Rosenstiel, Andreas Herkersdorf. Architecture and design methodology for autonomic system on chip 3rd 1183 colloquium in stuttgart. http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/projekte/oc/inhalte/inhalte/PresentationsStuttgart/04_presentation-ASoC.pdf, September 2006.

[24] Wolfgang Trumler, Faruk Bagci, Jan Petzold, Theo Ungerer, editor. *Smart Doorplate*. The First International Conference on Appliance Design (1AD), 2003.

[25] Wolfgang Trumler, Jan Petzold, Faruk Bagci, and Theo Ungerer, editor. *Towards an Organic Middleware for the Smart Doorplate Project*. GI Workshop on Organic Computing, September 2004.